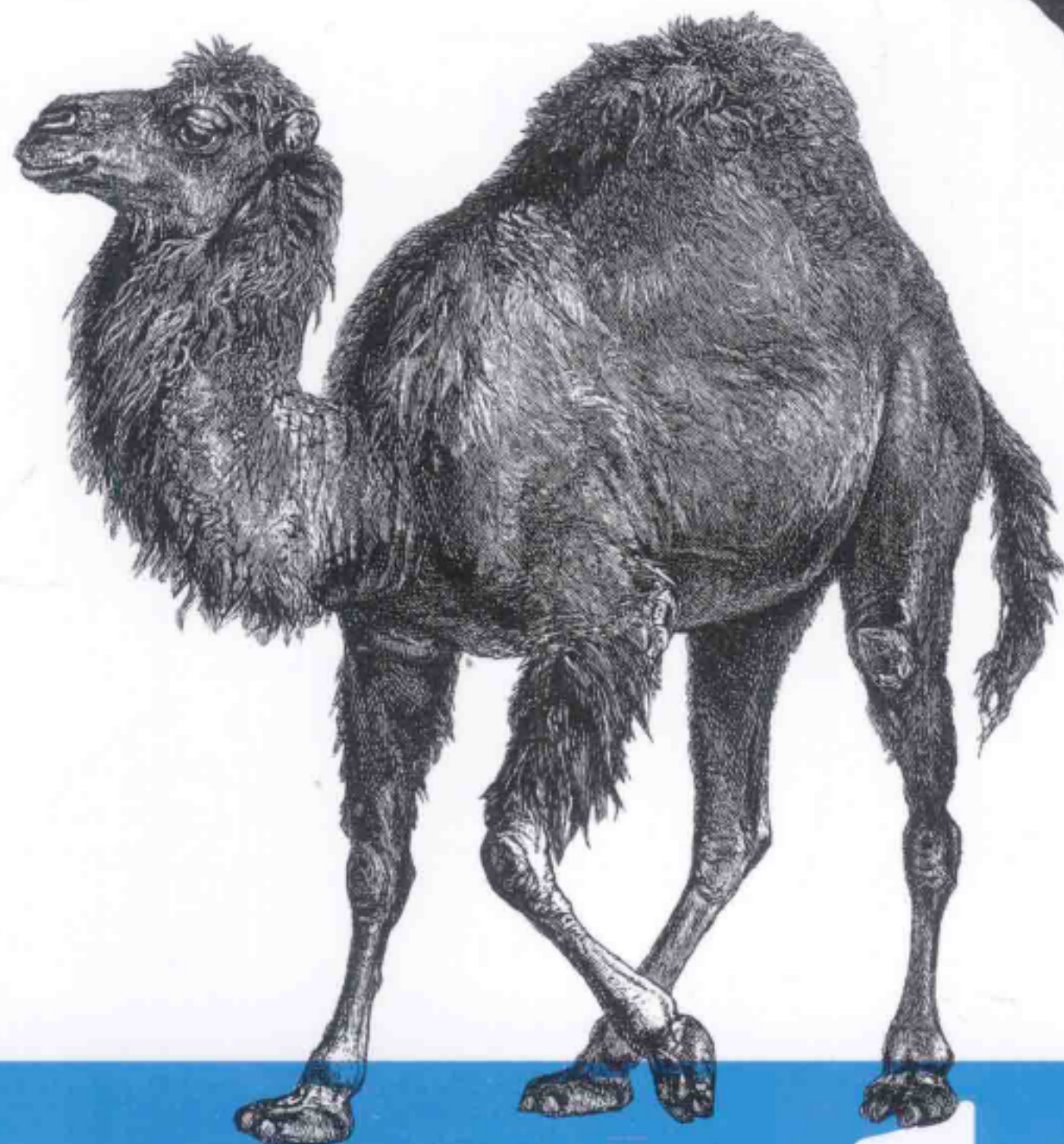


Programming Perl

第4版  
涵盖5.14



# Perl

## 语言编程

*Tom Christiansen,  
brian d foy & Larry Wall*

Jon Orwant 著  
苏金国 吴爽 等译

**O'REILLY®**  
中国电力出版社



# Perl语言编程

从1991年第一版问世以来,《Perl语言编程》很快成为无可争议的Perl宝典,如今仍是这种高实用性语言的权威指南。Perl最初只是作为一个功能强大的文本处理工具,不过很快发展成为一种通用的编程语言,可以帮助成千上万的程序员、系统管理员,以及像你一样的技术爱好者轻松完成工作。

人们早已经翘首以待这本“大骆驼书”的更新,如今终于得偿所愿。在这一版中,三位颇有声望的Perl作者讲述了这种语言当前版本(Perl 5.14)的内容,另外简要介绍了将来5.16版本中将要增加的一些特性。当今世界中,Unicode对于文本处理越来越重要,为此Perl提供了最好、最方便的支持,可以在任何地方平滑地集成Unicode,甚至在Perl最让人欢迎的特性(正则表达式)中也可以结合Unicode。

此次更新的重要特性包括:

- 新增的关键字和语法。
- I/O层和编码。
- 新增的反斜线转义。
- Unicode 6.0。
- Unicode字形簇和属性。
- 正则表达式中的命名捕获。
- 递归和文法模式。
- CPAN的扩展内容。
- 当前最佳实践。

Tom Christiansen是一位图书作者,同时也是一位Perl培训师,主要致力于文本挖掘、自然语言处理和计算语言学等领域。他合作撰写过《Perl Cookbook》和大量在线Perl文档。

brian d foy, Perl培训师和图书作者,合作编写了《Learning Perl》、《Intermediate Perl》和《Effective Perl Programming》。他还独立编写了《Mastering Perl》。

Larry Wall, Perl的创始人,也是Perl文化的主要倡导人,作为一位跨界艺术家,主要致力于将他的语言学和人类学经验应用到计算机科学中。

Jon Orwant创办了《The Perl Journal》,由于他对Perl的巨大贡献,于2004年被授予“白骆驼”终身成就奖。

**O'REILLY®**  
oreilly.com.cn

O'Reilly Media, Inc. 授权中国电力出版社出版

此简体中文版仅限于在中华人民共和国境内(但不允许在中国香港、澳门特别行政区和中国台湾地区)销售发行  
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-5123-5969-7



定价: 148.00元



第4版

---

# Perl语言编程

*Tom Christiansen, brian d foy  
& Larry Wall, Jon Orwant* 著  
苏金国 吴爽 等译

**O'REILLY®**

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社



## 图书在版编目 (CIP) 数据

Perl语言编程: 第4版/ (美) 克里斯蒂安森 (Christiansen, T.) 等著; 苏金国等译.  
—北京: 中国电力出版社, 2014.9

书名原文: Programming Perl, 4e

ISBN 978-7-5123-5969-7

I. ①P… II. ①克… ②苏… III. ①Perl语言—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字 (2014) 第116446号

北京市版权局著作权合同登记

图字: 01-2014-3144号

©2012 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2014.  
Authorized translation of the English edition, 2012 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc. 出版2012。

简体中文版由中国电力出版社出版2014。英文原版的翻译得到O'Reilly Media, Inc.的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc.的许可。

版权所有, 未得书面许可, 本书的任何部分和全部不得以任何形式重制。

封面设计/ Randy Comer, 张健  
出版发行/ 中国电力出版社 (www.cepp.sgcc.com.cn)  
地 址/ 北京市东城区北京站西街19号 (邮政编码100005)  
经 销/ 全国新华书店  
印 刷/ 北京丰源印刷厂  
开 本/ 787毫米×980毫米 16开本 60.625印张 1155千字  
版 次/ 2014年9月第一版 2014年9月第一次印刷  
印 数/ 0001-3000册  
定 价/ 148.00元 (册)

## 敬告读者

本书封底贴有防伪标签, 刮开涂层可查询真伪  
本书如有印装质量问题, 我社发行部负责退换

版权专有 翻印必究



# O'Reilly Media, Inc. 介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

## 业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal



# 译者序

这就是著名的大骆驼书！

人们把它称做是Perl的圣经，有意思的是，还有一些人认为这是一部“九阴真经”，而且是一部超厚的“九阴真经”！因为如果不用心体会，而一味贪多求快、囫囵吞枣，可能难以把握其中的妙处。不过，一旦读懂了这本书，你就能够在众多Perl程序员中脱颖而出，在错综复杂的环境中游刃有余。

关于Perl，在这里我们不打算过多赘述，毕竟，几位顶尖的Perl高手用了上千页来介绍这种语言，寥寥数语实在难以全面描绘它的形象，不过，可以肯定的是，Perl是一种能让你把事情做好的语言，已横跨众多操作平台，一旦上手，一定会让你爱不释手。

这本书的第三版于2000年问世，当时Perl v5.6还刚刚推出，时隔十多年，人们终于等来了这本大骆驼书的新版本，而那时Perl也已经演进到v5.14，实际上，在稍后的5月，Perl v5.16就正式发布了。如此长的跨度对于计算机图书特别是有关编程语言的图书来说确属少见。显而易见，这本大骆驼书的第四版会介绍大量新内容（实际上就是Perl的新内容），为了使你不至于应接不暇，本书对内容的组织和特性的介绍都做了大幅调整，具体有哪些变化？前言会为你细细道来。

你可能已经跃跃欲试，不过在翻开正文的第一页之前，先要有心理准备，这绝对不是一本通俗故事书，你需要静下心来，仔细看，慢慢看，还要多给自己一些练习的机会。有些内容甚至需要多读几次，反复推敲，才能茅塞顿开。在翻过一座又一座高山之后，你会发现你已经成为人们眼中的编程高手。

这本书的翻译历时一年多，其中艰辛自不必说。最关键的是，翻译这样一本巨著，确实让我们很有压力。所以我们不敢马虎，尽力用准确、贴切的语言表述出作者的原意。但是由于水平有限，肯定有不当之处，敬请批评指正。

——译者



## 作者介绍

---

**Tom Christiansen**是一位擅长Perl培训和写作的自由职业顾问。为TSR Hobbies（以其“龙与地下城”游戏而闻名）工作多年之后，他又返回大学，在西班牙待了一年，另外5年留在美国，热衷于音乐、语言学和编程，另外还学习了6种不同的语言。Tom最后终于从威斯康星大学-麦迪逊分校拿到了西班牙语和计算机科学的学士学位，另外还拿到计算机科学的硕士学位。接下来5年他在Convex担任一种万事通的角色，从系统管理到工具和内核开发都有涉足，另外还要负责客户支持和培训。Tom还在USENIX协会董事会担任了两届董事。由于有30多年Unix系统编程方面的经验，Tom举办过多场国际研讨会。如今Tom生活在科罗拉多州巨石市上面的丘陵地区，他的夏天总是在远足、骑马、捕鸟、作曲和游戏中心度过。

**brian d foy**是一个多产的Perl培训师和作家，他主办了“The Perl Review”，通过教育、咨询、代码审查等等帮助人们使用和了解Perl。他经常在Perl会议上发表演说。他是《Learning Perl》、《Intermediate Perl》和《Effective Perl Programming》的合作者，另外还单独著有《Mastering Perl》。1998年到2009年期间，他任职于Stonehenge Consulting Services担任讲师和作者。从他成为一个物理学研究生开始就是一个Perl用户，另外从他拥有自己的第一台计算机开始就是一个顽固的Mac用户。他成立了第一个Perl用户组（New York Perl Mongers），另外还创建了非盈利的Perl Mongers公司，帮助建立了全世界200多个Perl用户组。他维护着核心Perl文档的perlfaq部分，另外还维护着CPAN上的很多模块以及一些独立的脚本。

**Larry Wall**在Unisys当程序员的时候创建了Perl。现在他把全部时间用来指导这个语言的未来开发。Larry以其特殊的颇有创见的编程方法而闻名，另外也因他对自由软件编程文化做出的开创性贡献而享有盛誉。

Jon Orwant创办了《The Perl Journal》，并于2004年由于对Perl做出的卓越贡献而获得白骆驼终生成就奖。他是谷歌的项目经理，领导着Patent Search、可视化和数字人文科学小组。Jon在Google的大部分时间都参与Book Search的工作，他还建立了广泛使用的Google Books Ngram Viewer。在Google之前，他还曾是O'Reilly的CTO、France Telecom的研究部主任，另外还是MIT的讲师。Orwant于1999年得到了MIT电子出版组的博士学位。

## 封面介绍

---

本书的封面动物是一只单峰骆驼。单峰骆驼也称为“阿拉伯骆驼”，是一种单峰偶蹄反刍动物。单峰骆驼是骆驼家庭中体型最大的成员。它们作为家畜已经有3500年的历史，由于适应性很强，它们可以在沙漠里生存，这使它们成为非常理想的运输工具。

当今世界上的单峰骆驼主要是家畜，只有一种很著名的野生品种，生活在澳大利亚。家养的单峰骆驼生活在中东和北非，通常成群生活，一群中往往有多只雌性骆驼和一只雄性首领。单峰骆驼的驼峰可以储藏多达80磅脂肪，这可以分解为水和能量，使它即使没有水也可以在沙漠中跋涉100英里。除了能够在干旱中没有供给的条件下长途跋涉外，单峰骆驼还有两排睫毛，可以防止沙子进眼睛，另外遇到沙尘暴时还能把鼻孔合上，与马不同，它们屈膝来装载货物和人。单峰骆驼通常的寿命是40~50年。



# 目录

前言 .....	1
----------	---

## 第1部分 概述

第1章 Perl概述 .....	19
Perl入门 .....	19
自然语言和人工语言 .....	20
一个平均分例子 .....	32
文件句柄 .....	35
操作符 .....	38
控制结构 .....	44
正则表达式 .....	51
列表处理 .....	57
有些内容你不知道也没有（太大）危害 .....	59

## 第2部分 细节详述

第2章 集腋成裘 .....	63
原子 .....	63
分子 .....	64
内置数据类型 .....	66
变量 .....	67
名字 .....	69

标量变量 .....	73
上下文 .....	83
列表值和数组 .....	86
散列 .....	90
类型团和文件句柄 .....	91
输入操作符 .....	93
<b>第3章 一元和二元操作符 .....</b>	<b>99</b>
项和列表操作符（左边） .....	101
箭头操作符 .....	103
自增和自减 .....	103
指数 .....	104
表意一元操作符 .....	104
绑定操作符 .....	105
乘除操作符 .....	106
加减操作符 .....	107
移位操作符 .....	108
命名一元操作符和文件测试操作符 .....	108
关系操作符 .....	113
相等操作符 .....	113
智能匹配操作符 .....	114
C风格的逻辑（短路）操作符 .....	119
范围操作符 .....	121
条件操作符 .....	123
赋值操作符 .....	125
逗号操作符 .....	126
列表操作符（右边） .....	127
逻辑与、或、非和异或 .....	127
Perl中没有的C操作符 .....	128
<b>第4章 语句和声明 .....</b>	<b>129</b>
简单语句 .....	129
复合语句 .....	131
if和unless语句 .....	132



given语句 .....	133
循环语句 .....	138
goto操作符 .....	147
远古的Perl Case结构 .....	148
省略语句 .....	150
全局声明 .....	151
作用域声明 .....	153
Pragmas .....	160
<b>第5章 模式匹配 .....</b>	<b>163</b>
正则表达式家族 .....	164
模式匹配操作符 .....	166
元字符和元符号 .....	184
字符类 .....	193
位置 .....	205
分组与捕获 .....	208
候选项 .....	217
保持控制 .....	218
非传统模式 .....	230
<b>第6章 Unicode .....</b>	<b>255</b>
为什么会这样呢? .....	255
展示, 不要告诉 .....	259
获取Unicode数据 .....	261
张冠李戴 .....	265
字形和规范化 .....	267
Unicode文本比较和排序 .....	274
<b>第7章 子例程 .....</b>	<b>289</b>
语法 .....	289
传引用 .....	297
原型 .....	298
子例程属性 .....	306

<b>第8章 引用 .....</b>	<b>309</b>
什么是引用? .....	309
创建引用 .....	311
使用硬引用 .....	317
符号引用 .....	326
大括号、中括号和引号 .....	327
<b>第9章 数据结构 .....</b>	<b>332</b>
数组的数组 .....	332
数组的散列 .....	340
散列的数组 .....	342
散列的散列 .....	344
函数的散列 .....	347
更复杂的记录 .....	348
保存数据结构 .....	351
<b>第10章 包 .....</b>	<b>353</b>
符号表 .....	355
限定名 .....	358
默认包 .....	359
改变包 .....	360
自动加载 .....	362
<b>第11章 模块 .....</b>	<b>365</b>
加载模块 .....	366
上传模块 .....	367
创建模块 .....	368
覆盖内置函数 .....	374
<b>第12章 对象 .....</b>	<b>376</b>
面向对象术语简单回顾 .....	376
Perl的对象系统 .....	378
对象构造 .....	384
类继承 .....	388



实例析构造函数 .....	398
管理类数据 .....	406
Moose模块 .....	409
小结 .....	410
<b>第13章 重载 .....</b>	<b>411</b>
overload Pragma .....	412
重载处理方法 .....	412
可重载操作符 .....	414
复制构造函数(=) .....	421
缺少重载处理方法时 (nomethod和fallback) .....	422
重载常量 .....	423
公共重载函数 .....	424
继承和重载 .....	425
运行时重载 .....	425
重载诊断 .....	425
<b>第14章 绑定变量 .....</b>	<b>426</b>
绑定标量 .....	428
绑定数组 .....	436
绑定散列 .....	441
绑定文件句柄 .....	447
一个解除绑定小陷阱 .....	457
CPAN上的模块 .....	459

## 第3部分 Perl的技术

<b>第15章 进程间通信 .....</b>	<b>463</b>
信号 .....	464
文件 .....	469
管道 .....	476
System V IPC .....	483

<b>第16章 编译 .....</b>	<b>494</b>
Perl程序的生命周期 .....	495
编译代码 .....	496
执行代码 .....	501
编译器后端 .....	504
代码生成器 .....	504
字节码生成器 .....	505
代码开发工具 .....	506
先编译，后解释 .....	508
 <b>第17章 命令行接口 .....</b>	 <b>512</b>
命令处理 .....	512
环境变量 .....	529
 <b>第18章 Perl调试器 .....</b>	 <b>536</b>
使用调试器 .....	537
调试器命令 .....	539
调试器定制 .....	546
不被注意的执行 .....	550
调试器支持 .....	551
Perl性能测试 .....	553
 <b>第19章 CPAN .....</b>	 <b>559</b>
历史 .....	559
存储库之旅 .....	560
CPAN生态系统 .....	563
安装CPAN模块 .....	566
创建CPAN模块 .....	569
 <b>第4部分 Perl的文化</b>	
 <b>第20章 安全 .....</b>	 <b>577</b>
处理不安全的数据 .....	577
处理计时问题 .....	589



处理不安全的代码 .....	595
<b>第21章 常用实践 .....</b>	<b>604</b>
新手的常见失误 .....	604
效率 .....	615
有风格的编程 .....	624
老练的Perl .....	628
程序生成 .....	637
<b>第22章 可移植的Perl .....</b>	<b>641</b>
换行符 .....	643
字节顺序和数字宽度 .....	644
文件和文件系统 .....	645
系统交互 .....	646
进程间通信 (IPC) .....	647
标准模块 .....	648
日期与时间 .....	648
国际化 .....	648
风格 .....	649
<b>第23章 Pod .....</b>	<b>650</b>
Pod核心技术 .....	650
Pod转换器和模块 .....	658
编写自己的Pod工具 .....	659
Pod陷阱 .....	664
为Perl程序建立文档 .....	666
<b>第24章 Perl文化 .....</b>	<b>667</b>
历史决定成败 .....	667
Perl诗歌 .....	670
Perl程序员的品质 .....	672
大事记 .....	672
获得帮助 .....	673

## 第5部分 参考资料

<b>第25章 特殊名</b> .....	<b>677</b>
按类型分组的特殊名 .....	677
按字母顺序排列的特殊变量 .....	681
<b>第26章 格式</b> .....	<b>701</b>
字符串格式 .....	701
二进制格式 .....	707
形象格式 .....	717
<b>第27章 函数</b> .....	<b>724</b>
按类别组织的Perl函数 .....	727
按字母顺序组织Perl函数 .....	729
<b>第28章 标准Perl库</b> .....	<b>877</b>
标准库术语 .....	877
Perl库之旅 .....	879
<b>第29章 实现Pragma的模块</b> .....	<b>885</b>
attributes .....	886
autodie .....	887
autouse .....	887
base .....	888
bigint .....	889
bignum .....	890
bigrat .....	890
blib .....	890
bytes .....	891
charnames .....	891
constant .....	894
deprecate .....	896
diagnostics .....	897
encoding .....	899

feature.....	899
fields.....	900
filetest.....	900
if .....	900
inc::latest .....	901
integer.....	901
less .....	902
lib .....	902
locale .....	904
mro .....	904
open.....	905
ops .....	906
overload.....	906
overloading .....	907
parent.....	907
re .....	908
sigtrap.....	910
sort .....	912
strict .....	913
subs .....	915
utf8 .....	917
vars .....	917
version .....	918
vmsish .....	918
warnings .....	919
用户自定义Pragma .....	922
<b>术语表 .....</b>	<b>925</b>



# 前言

## 当幸福来敲门

Perl是一种能为你完成任务的语言。

当然，如果你的任务就是编程，理论上讲，可以用任何“完整”的计算机语言来完成任务。不过，从我们的经验来看，各种计算机语言之间的区别并不在于它们能够做什么，而很大程度上在于它们能不能容易地做到。在一个极端上，所谓的“第四代语言”可以很容易地做到某些事情，但用它们做另外一些事情则几乎不可能。另一个极端上，那些所谓的“工业强度级”语言做任何事情几乎都同样困难。

Perl有所不同。本质上讲，设计Perl的目标就是让简单的工作更容易，让困难的工作也并非遥不可及。

那么，这些应当很容易的“简单工作”是什么呢？当然就是你每天要做的那些工作。你需要一种语言，可以轻松的处理数字、文本、文件和目录、计算机和网络，特别是程序。它应该能很容易地运行外部程序，检查输出，找到有意思的东西。能够把这些有意思的东西交给其他程序，让它们做一些特殊处理。还应该能很容易地开发、修改和调试你自己的程序。当然，要能以一种可移植的方式在任何现代操作系统上编译和运行你的程序。

所有这些Perl都能做到，而且还远不止这些。

Perl最初是为UNIX设计的一种黏合语言，经过不断发展，早已经延伸到大多数其他操作系统。由于Perl几乎可以在任何地方运行，这也是当前最具移植性的编程环境之一。如果想编写可移植的C或C++程序，必须针对不同的操作系统在程序上加上一大堆奇怪的#ifdef标记。要想编写可移植的Java程序，你必须了解每一个新Java实现的种种特异之处。要编写

一个可移植的shell脚本，由于命令在不同操作系统上会有不同的版本，就得记住每一条命令在每一个操作系统上的语法，并以某种方式找出普遍适用（希望如此）的共同要素。要编写可移植的Visual Basic程序，则需要对“可移植”一词有更灵活的定义。

让人高兴的是，Perl很好地避免了这些问题，同时保留了这些语言的许多优点，还具有自己独有的一些魔力。Perl的魔力来自很多方面：Perl特性集提供的工具、Perl社区的创造性，以及开源运动的大环境。不过，这些魔力大多是杂合带来的优势；Perl得到的是“混合遗产”，对Perl而言，多样性是一种优点，而非缺点。Perl可算是一种“解救性”的语言，就像自由女神像基座上刻下的“把你的疲累，你的困苦，统统交给我。”如果你感觉身陷繁乱，渴望自由，那么请使用Perl。

Perl跨越了多种文化。前UNIX系统程序员的热切期望促成了Perl如今的爆炸性增长，他们希望尽可能地保留“故国”原有的东西。对他们来说，Perl是对UNIX文化的可移植升华，就像“此路不通”的荒漠中的一个绿洲。另一方面，Perl在另一个方向上也大有作为：基于Windows的Web设计人员通常很高兴地发现，他们的Perl程序可以不做任何改动就能直接在公司的UNIX服务器上运行。

Perl在系统程序员和Web开发人员中尤其流行，这是因为他们最早发现了Perl；不过，Perl还有更广泛的用途。Perl最早只是一种文本处理语言，后来发展成为一种复杂、通用的编程语言，提供了丰富特性的软件开发环境，包括调试工具、性能分析工具、源码分析工具、编译器、库、可以指示语法的编辑器，以及“真正的”编程语言拥有的所有其他工具（如果你需要）。不过，所有这些都是为了解决难题，而且很多语言都能做到这一点。Perl的独特之处在于，它永远没有忘记要保证很容易地完成简单的工作。

由于Perl不仅功能强大，而且简便易用，所以被广泛应用于各个领域，从宇航工程到分子生物学，从数学到语言学，从图形处理到文本处理，从数据库操作到客户服务器网络管理。很多人迫切需要快速地分析或转换大量数据，比如说DNA序列、Web页面或者其他意想不到的东西，碰到这些问题时他们就会使用Perl。

Perl的成功有很多原因。早在开源运动兴起之前，Perl已经是一个成功的开源项目了。Perl是免费的，而且会一直免费。你可以采用你认为合适的任何方式使用Perl，只需要遵守一个很宽松的许可条款。如果你在创业，想用Perl，没问题，可以放手使用。可以把Perl嵌在你编写的商业应用中，无需支付费用，也没有任何限制。如果你遇到一个问题，而Perl社区无法解决，那么还有最后一招：可以查看源代码。Perl社区不会借“升级”之名告诉你商业秘密。另外Perl社区永远不会“停业”，不会让你孤立无援。

作为一款免费软件，这对Perl的发展无疑是有帮助的。不过，这还不足以解释Perl现象，因为还有很多软件包是免费的，却未能发展起来。人们觉得在Perl的世界里很有创造力，因为他们可以自由地表达：可以选择针对什么进行优化，可能是计算机速度或者程序员速



度，也可能是详细或者简洁，或者是可读性、可维护性、可重用性、可移植性、易学性或易教性。如果你在参加一个模糊Perl大赛，甚至可以针对模糊性进行优化。

Perl可以给予你所有这些自由，因为它就像有一种“分裂人格”。Perl是一种非常简单的语言，同时也是一种相当丰富的语言。Perl从各个方面汲取好的想法，并置于一个易于使用的指导框架中。对于那些有点喜欢它的人来说，Perl是实用摘录与报表语言（Practical Extraction and Report Language）。而对于那些真正热爱它的人而言，Perl则是病态折中垃圾列表器（Pathologically Eclectic Rubbish Lister）。对于极简主义者，Perl看来就像无意义的冗余练习。不过没关系，这个世界还是需要一些精简主义者的（主要是物理学家）。精简主义者喜欢分解事物，而我们其余的人则着力实现组合。

在很多方面Perl都是一种简单的语言。你不用知道多少特殊指令就能编译一段Perl程序，可以像执行一个批处理文件或shell脚本一样执行Perl程序。Perl的类型和结构很容易使用，也很好理解。对于你使用的数据，Perl未施加任何限制，字符串和数组要多大就可以扩展到多大（只要你有足够的内存），而且有很好的可扩展性。Perl并不强迫你学习新的语法和语义，它的很多特性都借用自你很熟悉的其他语言（例如C、awk、BASIC、Python、英语和希腊语等）。实际上，几乎每一个程序员都能读懂一段编写良好的Perl代码，知道它要做什么。

最重要的是，在编写有用的程序之前，你不必全面了解有关Perl的一切。你可以先学一点点Perl。可以用Perl写一些幼稚的程序（我们保证不会笑话你）。更确切地说，就像不会笑话小孩子的创意行为一样，我们绝对不会笑话你的Perl作品。Perl的很多思想都借鉴于自然语言，其中最棒的一点是，只要能把自己的意思表达清楚，完全可以只使用语言的一个子集。在Perl文化中，任何熟练程度都可以接受。我们不会叫语言警察来纠正你。如果你的Perl脚本能完成工作（不会让你的老板因此“炒掉”你），它就是正确的脚本。

尽管在很多方面很简单，但Perl同时也是一种相当丰富的语言，有很多东西需要学习。这也是解决难题需要付出的代价。要想真正掌握Perl的所有功能，这确实需要花费一些时间，不过这是值得的，等你需要完成某些工作时，将会很高兴地发现完全可以利用Perl的丰富功能来做到。

由于Perl继承了很多语言的特性，所以，即使当它“只”是一种数据归约语言时也有丰富的特性，可以用来实现文件导航、扫描大量文本、创建和获得动态数据，以及基于这些数据轻松地打印格式化报表。不过，不知从何时起，Perl开始流行起来。它成为一种有多种用途的语言，可以实现文件管理系统、进程管理、数据库管理、客户服务器编程、安全编程、基于Web的信息管理，甚至还可以用于面向对象编程和函数编程。这些功能并非简单地在Perl中罗列，每个新功能都能很好地与其他功能协作，因为Perl从一开始就设计为一种黏合语言。

不过Perl不仅仅能黏合自己的特性。Perl语言设计为一种模块化语言，可以很好地扩展。



使用Perl，你可以快速设计、编写、调试和部署应用，还可以随着需求的增加轻松地扩展这些应用的功能。可以把Perl嵌在其他语言中，也可以将其他语言嵌在Perl中。通过模块导入机制，可以方便地使用这些外部功能定义，就像是Perl的内置特性一样。面向对象外部库在Perl中仍然保持其面向对象性。

Perl在其他方面也会有帮助。与严格的解释型语言不同（如命令文件或shell脚本），那些语言编译和执行程序时一次只处理一条命令，而Perl首先将整个程序快速编译为一种中间格式。与其他编译器一样，它会完成各种优化，并立即提供各种编译反馈，指出语法和语义错误，以及库绑定出现的问题。一旦Perl编译器前端通过了你的程序，就将编译得到的中间代码交给解释器来执行（或者也可以交给任何能执行C或字节码的模块化后端来执行）。听上去很复杂，不过编译器和解释器效率都很高，通常我们发现一般的“编译—运行—修改”周期只需几秒就能完成。结合Perl的很多故障弱化（fail-soft）特性，Perl的这种快速功能使之成为建立快速原型的理想语言。建立原型之后，随着程序的逐步成熟，可以慢慢拧紧螺钉，减少程序中的散漫成分，让它“更守纪律”。如果适当地进行要求，Perl也能帮你这个忙。

Perl还可以帮你编写更安全的程序。除了其他语言提供的那些典型的安全接口外，Perl还通过一个特有的数据跟踪机制来避免意外的安全错误，可以自动确定哪些数据来自非安全来源，并避免有危险的操作发生。最后一点，Perl允许你建立特别的保护区，可以在这里安全地执行那些来历不明的Perl代码，隔离有危险的操作。

不过，有些矛盾的是，Perl对你最有帮助的反而与Perl本身没有太大关系，而是与使用Perl的人有关。坦率地说，Perl社区的人是世界上最热心的人。如果Perl运动有一些宗教性，那么这就是它的核心。Larry希望Perl社区像一个小天堂，到目前为止，大体说来看上去他已经实现了这个愿望。也请尽你之力守护这个小天堂吧。

不论你学习Perl的原因是什么，也许是想拯救地球，或者只是感到好奇，也可能是你的老板要求你这么做，你将从这本书找到你想要的，它会带着你由浅入深地了解基本知识和疑难问题。尽管我们不打算教你如何编程，不过有敏锐观察力的读者肯定能从中发现一些编程的艺术，掌握一点编程的科学。我们会鼓励你培养程序员的三种“优秀品质”：懒惰、急躁和傲慢。在这个过程中，希望你能发现这本书中有些地方有点意思（其他地方更是极其有趣）。如果这些还不能让你保持清醒，请不断提醒自己，学习Perl可以增加你的简历的分量。所以坚持读下去吧。

## 本版新内容

也许应该反过来讲，这一版哪些不是新内容？这本书上一次更新至今已经过去很长时间了。在此期间我们遇到了一些问题，所以没能及时做出更新，不过现在情况好多了，终于能把这本书的更新摆上日程。

第三版是2000年中期出版的，那时Perl v5.6刚刚推出。而现在已经十多年之后，Perl v5.16都很快要问世了。这些年间发生了很多事情，包括Perl 5推出了很多新版本，还出现了一个Perl 6。不过，这个6有欺骗性；Perl 6实际上只是Perl 5的“姊妹”语言，它并不是Perl 5的全面更新，不要因为这个版本号而期待过高。这本书并不介绍Perl 6，我们讨论的仍然是Perl 5，这也是目前全世界大多数人（甚至包括使用Perl 6的人）熟悉的版本，而且仍在大量使用。<sup>注1</sup>

要指出这本书中有哪些新内容，实际上就是要告诉你Perl有哪些新增的内容。这可不是为了增加这本书的销售额而简单地提供一个新版本，只是小打小闹地做些调整。Perl在过去五年已经相当活跃，人们对这本书的全面更新早已翘首以待。我们不会列出哪些有改变（有关内容可以查看perldelta页面，其中会指出这一版本与上一版本的区别），不过有些内容我们会特别强调。

在Perl 5中，我们开始加入一些重要的新特性，并提供了一种方法允许老程序保持原来的风格，而不必换上新关键字。例如，由于大家强烈要求增加一个类似switch的语句，最后我们终于让步了。不过，通过采用典型的Perl方式，这个新特性可以做得更好、更出色，使你能更好地控制要做些什么。我们把它叫做given-when语句，不过只有当你做出要求时才会得到这个特性。以下语句都能启用这个特性：

```
use v5.10;
use feature qw(switch);
use feature qw(:5.10);
```

一旦启用，你就拥有了这个“超大能量的switch”：

```
given ($item) {
    when (/a/) { say "Matched an a" }
    when (/bee/) { say "Matched a bee" }
}
```

有关的更多内容将在第4章介绍，另外还会在合适的地方介绍其他一些新特性。

尽管Perl从v5.6开始就已经提供Unicode支持，不过这项支持在最近几个版本中又得到了显著改进，这包括与同期所有其他语言相比，它能提供更好的正则表达式支持。由于Perl不断提供越来越好的支持，这使它成为未来Unicode开发的一个试验平台。在这本书的前几个版本中，与Unicode有关的内容都放在一章中介绍，不过这一版有所不同，全书中都会看到它的身影（只要需要，我们就会介绍）。

很多人一提到正则表达式就会与Perl关联起来，实际上现在Perl的正则表达式甚至比以前更加出色。其他语言参考了Perl的模式语言，把它称作Perl兼容正则表达式（Perl

---

注1：我们有点懒，另外也因为现在你已经知道这本书只介绍Perl 5，所以这里特别说明，这本书后面我们并不总会完整地写出版本“Perl v5.n”，如果你看到“v5”开头的版本号，就应该想到我们讨论的是Perl的这个版本。



Compatible Regular Expressions)，另外还增加了它们自己的一些特性。我们又汲汲了其中一些特性，继续将Perl集众家所长的传统发扬光大。你会发现一些强大的新特性，可以用来处理模式中的Unicode。

如今线程也大不相同了。之前Perl支持两个线程模型：一个名为5005threads（因为这是发布Perl v5.005版本时加入的），另一个是解释器线程（interpreter threads）。从v5.10开始只支持解释器线程。不过，出于多种原因，我们并不认为这一版能很好地解释这个内容，因为还要花大量时间讨论其他的一些特性。如果你希望更多地了解线程，可以看看perlthrtut手册页（manpage），假如我们专门有一章介绍“线程”，应该也与这个手册页的内容相差无几。不过，也许以后我们会另外增加一章作为“奖励”。

另外还有一些或增加或去除的特性。有些试验不成功，所以我们把那些特性“请出”了Perl，取而代之以另外一些试验。例如，伪散列（Pseudohashes）就已经过时并被删除，而且已经被我们淡忘。如果你不知道它是何物，不用担心，不过也别指望这一版会介绍该内容。

另外，自这本书最近一次更新以来，Perl编程实践以及相应的测试文化发生过一次（也可以说是两次）翻天覆地的变化。CPAN（Perl综合典藏网，Comprehensive Perl Archive Network）上的内容继续呈指数增长，使之成为Perl的突出特性。尽管这不是一本专门介绍CPAN的书，不过我们会在必要时介绍一些模块。不要试图用最基本的Perl搞定一切。

我们还删去了两章：标准库模块清单（上一版的第32章）和诊断消息（上一版的第33章）。可能在你将这本书摆上书架之前这两章就已经过时了。我们会教你如何自己找到这个模块清单。至于诊断消息，完全可以在perldiag手册页中找到，也可以用diagnostics pragma把警告转换为更长更详细的消息。

## 第1部分 概述

迈出第一步总是最艰难的。这一部分会用一种轻松随意的方式介绍Perl的基本概念。它并不是一个面面俱到的教程，而只是帮你快速入门，也许不能满足所有人的需要。如果你不适应这种学习方式，想找其他更合适的书，可以参考后面提供的“离线文档”一节。

## 第2部分 细节详述

这一部分从各个抽象层次对这种语言进行了全面、深入的讨论，从数据类型、变量和正则表达式到子例程、模块和对象都有介绍。你将对这个语言如何工作有更好的认识，在学习过程中，你还能得到有关设计优秀软件的一些经验（如果你从未用过支持模式匹配的语言，这一部分还会给你一些额外的“好处”）。

## 第3部分 Perl的技术

Perl本身可以做很多事情，不过这一部分将让你更上一个层次。在这里你将了解如何让Perl跳出计算机设置的诸多“圈子”，这包括方方面面，从处理Unicode、进程间



通信和多线程，到编译、调用、调试和Perl性能测试，以至用C或C++编写你自己的外部扩展，或者与你喜欢的现有API建立接口。Perl很乐于与你的计算机（或者说可以这么说，只要得到允许，可以是互联网上的任何计算机）上的任何接口交互。

#### 第4部分 Perl的文化

每个人都知道，一种文化必须有自己的一种语言，不过Perl社区总认为一种语言必须有一种文化。在这一部分中，我们将把Perl编程看作是嵌入在现实世界中的一种人类活动。我们会介绍如何改善与好人和坏人的相处。另外还会给出很多建议，告诉你如何让自己成为更出色的人，以及如何让你的程序对别人更有用。

#### 第5部分 参考资料

这一部分中，我们将各章的内容汇集在一起，你可能希望按字母顺序从中查找你想要的东西，包括特殊变量和函数，以及标准模块和pragma等内容。对于那些不太熟悉计算机科学术语的人来说，术语表会非常有用。例如，如果你不知道“pragma”的含义，现在就可以在术语表中找到（当然，如果你连“is”是什么意思都不知道，那我们就帮不了你了）。

## 标准发布版本

官方Perl策略（参见perlpolicy）要求正式支持最近的两个维护版本。由于写这本书时的当前版本是v5.14，这意味着得到正式支持的版本包括v5.12和v5.14。等到发布v5.16版本时，v5.12将不再支持。

如今，大多数操作系统开发商都把Perl当作系统的一个标准组件，不过这些操作系统发布新版本时可能未能涵盖最新版本的Perl。写这本书时，AIX、BeOS、BSDI、Debian、DG/UX、DYNIX/ptx、FreeBSD、IRIX、LynxOS、Mac OS X、OpenBSD、OS390、RedHat、SINIX、Slackware、Solaris、SuSE和Tru64都提供了Perl，作为其标准版的一部分。有些公司在单独的免费软件光盘上提供Perl，或者通过各自的客户服务组来提供。一些第三方公司（如ActiveState）为多种不同操作系统提供了预编译的Perl版本，包括Microsoft公司的各个操作系统。

即使你的操作系统开发商确实在标准版本中提供了Perl，你可能还是希望自己来编译和安装Perl。这样一来，你可以确信有最新的版本，而且能够选择库和文档的安装位置。你还能选择编译Perl时是否支持一些可选的扩展包，如多线程、大文件或通过-D命令行开关支持多个底层调试选项（用户级Perl调试器肯定是支持的）。

要下载一个Perl源工具箱，最容易的办法可能就是通过Web浏览器访问Perl主页，在启动页面上可以找到很醒目的下载信息，另外主页上还提供了一些链接，它们指向面向多种平台的多个预编译库（这些平台没有合适的C编译器）。

还可以直接访问第19章介绍的CPAN (<http://www.cpan.org>)。如果你觉得太慢(可能确实很慢,因为访问的人很多),可以找到一个离你比较近的镜像。它会提供一个MIRRORED.BY文件,其中包含所有其他CPAN网站的列表。可以找到这个文件,然后选出你最喜欢的镜像。有些可以通过FTP得到,还有一些需要通过HTTP(这对于一些公司的防火墙会有一点差别)。<http://www.cpan.org>多路复用器会为你做这个选择。如果你愿意,以后还可以改变你的选择。

一旦得到源代码,并解压到一个目录中,接下来应该读一读其中的README和INSTALL文件,学习如何构建Perl。可能还包括一个INSTALL.platform文件供你阅读,这里的platform表示你的操作系统平台。

如果你的操作系统平台(platform)恰好是UNIX系列的某个版本,获取、配置、构建和安装Perl的命令将与下面的命令很类似。首先,必须选择一个命令来获取源代码。可以使用一个浏览器或命令行工具通过Web下载:

```
% wget http://www.cpan.org/src/5.0/maint.tar.gz
```

现在解压缩,并完成配置、构建和安装:

```
% tar xzf latest.tar.gz      # 或者先gunzip, 再tar xf
% cd perl-5.14.2             # 或 5.* 对应任何版本号
% sh Configure -des           # 假设缺省答案
% make test && make install    # 安装通常要求是超级用户
```

你的平台可能已经提供了一些安装包,可以为你完成这个工作(并提供平台特定的补丁或增强包)。甚至很多平台已经提供了Perl,所以你可能无须做任何工作。

如果你已经安装了Perl,不过想要一个不同的版本,可以使用perlbrew工具,它会让你少做很多工作。这个工具会为你自动完成Perl的构建,把它安装在你指定的某个位置(只要你在这个目录上有安装文件的权限,而不需要任何管理员特权)。在CPAN上这个工具是App::perlbrew,不过,也可以根据文档来安装:

```
% curl -L http://xrl.us/perlbrewinstall | bash
```

一旦安装,就可以让这个工具为你完成所有工作:

```
% ~/perl5/perlbrew/bin/perlbrew install perl-5.14.2
```

perlbrew还能为你做更多事情,所以请查阅文档。

还可以得到增强版本的标准Perl发布。ActiveState为Windows、Mac OS X和Linux免费提供了ActivePerl,不过为Solaris、HP-UX和AIX提供的ActivePerl需要收取一定费用。

Strawberry Perl只面向Windows,它提供了为CPAN编译和安装第三方Perl模块所需的各种工具。



Citrus Perl是一个适用于Windows、Mac OS X和Linux的版本，内置有wxPerl工具，可以用来创建GUI。这个版本面向那些想用Perl（而非通用Perl）创建分布式GUI应用的人。它的Cava Packager工具会帮助你达到目的。

## 在线文档

Perl提供了大量在线文档作为标准Perl版本的一部分（可以参阅下一节来了解离线文档）。从CPAN安装模块时还会显示另外的文档。

这本书中提到“Perl手册页”时，就是指你的计算机中保存的一组在线Perl手册页面。手册页（manpage）一词只是一个习惯说法，表示一个包含文档的文件，阅读这个文件并不需要一个UNIX man程序。甚至可以将Perl手册页安装为HTML页面，特别是在非UNIX系统上。

Perl的在线手册页划分为几个独立的部分，以便你找到想要的东西，而不用翻阅数百个文本页面。因为顶级手册页就叫做Perl，执行UNIX命令“`man perl`”就能看到这个顶级手册页<sup>注2</sup>。这个页面会带你进入更具体的页面。例如，“`man perlre`”会显示Perl正则表达式的手册页。不能使用man命令的系统上通常可以使用`perldoc`命令。你的系统可能还提供HTML格式的Perl手册页，或者采用系统特有的某种帮助格式。请咨询本地的系统管理员（除非你自己就是系统管理员）。不管怎样，有问题都可以找<http://perlmonks.org>。

## 查阅标准手册页

从一开始（当然是Perl开始的时候，这要追溯到1987年），Perl手册页是一个很简单的文档，排版和打印后大约有24页。例如，关于正则表达式的一节只有两段（如果你知道`egrep`，这就足够了）。在那之后，几乎所有一切都以某种方式发生了变化。算上标准文档、各种实用工具、各个平台的移植信息，再加上大量标准模块，如今，文档排版后已经有数千页，分为多个单独的手册页（这还没有算上你安装的CPAN模块，而且这可能也不少）。

不过，另外一些方面没有发生任何变化：Perl手册页依然存在。如果你不知道从哪里开始，Perl手册页仍是一个很好的起点。区别在于，一旦进入这个手册页，你就无法停步了。Perl文档不再是一个家庭小作坊，现在它已经是拥有数百家商店的超级购物中心。进入大门时，你要找到“你现在的位置”（YOU ARE HERE），确定哪家店出售你想买的东西。当然，一旦你熟悉了这个购物中心，就能直接知道该去哪里。

---

注2：如果执行这个命令时还是得到一个超大的页面，这可能是老版本的v4手册页。检查你的MANPATH，查看是否包含老版本的网站（可以执行“`perldoc perl`”，了解如何根据“`perl -V:man.dir`”的输出配置你的MANPATH）。



表P-1给出了你可能看到的几个“商店招牌”。

表P-1：部分Perl手册页

手册页	内容
<i>perl</i>	有哪些Perl手册页
<i>perldata</i>	数据类型
<i>perlsyn</i>	语法
<i>perlop</i>	操作符和优先级
<i>perlre</i>	正则表达式
<i>perlvar</i>	预定义变量
<i>perlsub</i>	子例程
<i>perlfunc</i>	内置函数
<i>perlmod</i>	perl模块如何工作
<i>perlref</i>	引用
<i>perlobj</i>	对象
<i>perlipc</i>	进程间通信
<i>perlrun</i>	如何支持Perl命令（以及命令行开关）
<i>perldebug</i>	调试
<i>perldiag</i>	诊断消息

这里只选择了一小部分，不过这也是很重要的一部分。可以看到，如果你想了解操作符，*perlop*很可能有你想要的东西。如果你想查找关于预定义变量的某个内容，可以查看*perlvar*。如果你得到一个看不懂的诊断消息，可以向*perldiag*寻求帮助。依此类推。

标准Perl手册中还有一部分是常见问题（Frequently Asked Questions, FAQ）清单。它分为以下9个不同页面，如表P-2所示。

表P-2：perlfaq手册页

手册页	内容
<i>perlfaq1</i>	关于Perl的一般问题
<i>perlfaq2</i>	获取和学习Perl
<i>perlfaq3</i>	编程工具
<i>perlfaq4</i>	数据操纵
<i>perlfaq5</i>	文件和格式
<i>perlfaq6</i>	正则表达式
<i>perlfaq7</i>	一般的Perl语言问题
<i>perlfaq8</i>	系统交互
<i>perlfaq9</i>	网络

有些手册页包含平台特定的说明信息，如表P-3所示。

表P-3：平台特定的手册页

手册页	内容
<i>perlamiga</i>	Amiga移植
<i>perlcygwin</i>	Cygwin移植
<i>perldos</i>	MS-DOS移植
<i>perlhpx</i>	HP-UX移植
<i>perlmachten</i>	Power MachTen移植
<i>perlos2</i>	OS/2移植
<i>perlos390</i>	OS/390移植
<i>perlvms</i>	DEC VMS移植
<i>perlwin32</i>	MS-Windows移植

（关于移植的有关信息，请参见第22章和前面介绍的CPAN ports目录。）

## 非Perl手册页

提到非Perl文档时，如`getitimer(2)`，这是指《Unix程序员手册》第2部分中的`getitimer`手册页<sup>注3</sup>。非UNIX系统上可能并不提供syscalls（如`getitimer`）的手册页，不过这也没有关系，因为非UNIX系统上根本不能使用UNIX syscall。如果你确实需要一个UNIX命令、syscall或库函数的文档，很多机构已经把它们的手册页放到了网上，在Google上快速搜索`crypt(3) manual`就能找到很多副本。

尽管顶级Perl手册页通常都安装在标准man目录的第1部分，不过本书中省掉了本该在这些手册页名后面追加的(1)。你肯定能认出它们，因为这些手册页的名字都采用“*perl\*\**”形式。

## 离线文档

如果你想更多地了解Perl，下面是我们推荐的一些相关的出版物：

- 《Perl 5 Pocket Reference》，Johan Vromans编著（O'Reilly出版，第5版，2011年7月）。这本小册子可以作为Perl的一个很方便的速查参考。

---

注3：第2部分只包含对操作系统的直接调用（这些通常称为“系统调用”，不过这本书中我们会把它们称为syscalls，以避免与“系统函数”相混淆，系统函数与syscalls毫无关系）。不过，对于哪些调用实现为syscalls，而哪些实现为C库调用，不同的系统会稍有区别，所以可以想象，可能会在第3部分找到`getitimer(2)`。

- 《Perl Cookbook》，Tom Christiansen和Nathan Torkington编著（O'Reilly出版，第2版，2003年8月）。这是你手上这本书的姊妹篇。这本“烹饪”手册中的技巧会教你如何用Perl“烹制美食”。
- 《Learning Perl》，Randal Schwartz、brian d foy和Tom Phoenix编著（O'Reilly出版，第6版，2011年6月）。这本书会教程序员70%的时间里要用到的30%的基本Perl内容，它主要面向那些编写自包含程序（大约几百行代码）的人。
- 《Intermediate Perl》，Randal Schwartz、brian d foy和Tom Phoenix编著（O'Reilly出版，2006年3月）。这本书介绍了《Learning Perl》未涵盖的内容，包括引用、数据结构、包、对象和模块。
- 《Mastering Perl》，brian d foy编著（O'Reilly出版，2007年6月）。这本书与《Learning Perl》和《Intermediate Perl》堪称三部曲，这是其中的最后一部。书中并非强调Perl语言的基础知识，而转向另一个角度，指导Perl程序员如何应用Perl完成实际工作。
- 《Modern Perl》，chromatic编著（Oynx Neon出版社，2010年10月）。这本书全面调查了现代Perl编程实践和相关主题，非常适合那些对编程已经有所了解但尚未关注Perl最新开发的人。
- 《Mastering Regular Expressions》，Jeffrey Friedl编著（O'Reilly出版，第3版，2006年8月）。尽管它没有涵盖Perl正则表达式增加的最新特性，不过如果想了解正则表达式是如何工作的，这绝对是一个非常宝贵的参考资源。
- 《Object Oriented Perl》，Damian Conway编著（Manning出版社，1999年8月）。对于初学者以及高级面向对象程序员，这本书解释了用Perl编写强大的对象系统时可采用的一些常用以及高深的技术。
- 《Mastering Algorithms with Perl》，Jon Orwant、Jarkko Hietaniemi和John Macdonald编著（O'Reilly出版，1999年）。这本书提供了计算机科学算法课程要介绍的所有有用的技术，不过不包括费劲的证明。这本书涵盖图形、文本、集合等领域的很多基本而实用的算法。

除此之外，还有很多其他的Perl书和出版物，无疑我们还有遗漏，可能还有一些很好的书没有提到（出于善意，我们也没有提到一些比较糟糕的书）。

除了以上所列与Perl相关的出版物，下面这些书并非与Perl直接相关，不过对于参考、咨询和提供灵感来说会大有帮助。

- 《The Art of Computer Programming》，Donald Knuth编著，卷1–4A：“基本算法”、“半数值算法”、“排序和搜索”以及“组合算法”（Addison-Wesley出版社，2011年）。



- 《Introduction to Algorithms》，Thomas Cormen、Charles Leiserson和Ronald Rivest 编著（MIT Press和McGraw-Hill出版社，1990年）。
- 《Algorithms in C》，Robert Sedgewick编著（Addison-Wesley出版社，1990年）。
- 《The Elements of Programming Style》，Brian Kernighan和P.J. Plauger编著（Prentice Hall出版社，1988年）。
- 《The UNIX Programming Environment》，Brian Kernighan和Rob Pike编著（Prentice Hall出版社，1984年）。
- 《POSIX Programmer's Guide》，Donald Lewine编著（O'Reilly出版社，1991年）。
- 《Advanced Programming in the UNIX Environment》，W. Richard Stevens编著（Addison-Wesley出版社，1992年）。
- 《TCP/IP Illustrated》，W. Richard Stevens编著，卷I-III；（Addison-Wesley出版社，1992年~1996年）。
- 《The Lord of the Rings》，J. R. R. Tolkien编著；美国的Houghton Mifflin出版社和英国的Harper Collins出版社（最后一次印刷时间：2005年）。

## 其他资源

互联网真是一个伟大的发现，我们还在努力发掘如何将它用到极致（当然，有些人更希望像托尔金发现中土世界一样“发现”互联网）。

## Perl网上资源

访问Perl网站。可以从中了解Perl世界的新闻，其中包含源代码和移植版本、专题文章、文档、会议日程以及很多其他内容。

另外可以访问Perl Mongers网页（<http://www.pm.org>），从更底层的角度了解Perl的基础，就像草根一样，在全世界除南极以外的任何地方都能茂盛生长（在南极，小草只能在室内生长）。本地PM组会定期召开小型会议，可以在这里与同一地区的其他Perl人员交流有关Perl的经验和知识。

## Bug报告

尽管几率不大，不过你确实有可能遇到Perl本身而不是你自己程序的bug，如果是这样，要尽量缩减为一个最小测试用例，然后用Perl提供的perlbug程序提交报告。有关的更多信息请参见<http://bugs.perl.org>。

`perlbug`命令实际上是RT bug跟踪工具<sup>注4</sup>实例的一个接口。如果不利用这个工具，也可以很容易地通过Email向`perlbug@perl.org`提交你的报告，不过`perlbug`会收集安装过程中的各项信息，如版本和编译选项，这些可以帮助Perl开发人员明确你遇到的问题。

还可以查看当前问题列表，因为可能有人以前遇到过同样的问题。先从<https://rt.perl.org/>开始，沿着perl5队列的相关链接追踪下去。

如果你在处理CPAN的一个第三方模块，就要用<https://rt.cpan.org/>上一个不同的RT实例。不过，并不是每一个CPAN模块都能使用免费的RT账户，所以请查阅模块的文档，了解报告bug的特别要求。

## 本书约定

有些约定本身就成为一个比较大的章节。编码约定将在第21章“有风格的编程”一节中讨论。从某种意义上讲，术语表给出了我们的词法约定。

本书使用了以下排版约定：

小体大写字母 (SMALL CAPITALS)

主要用于Unicode字符的正式名，另外也用于布尔操作符。

斜体 (*Italic*)

用于URL、手册页、路径名和程序。新术语在正文中第一次出现时也会用斜体。如果正文中的术语定义对你不适用，很多术语在术语表中还会有其他定义。另外斜体还用于命令名和命令行开关。这样有助于人们区分开关和操作符，例如，可以从字体上区别`-w`警告开关和`-w`文件测试操作符。

常规等宽字体 (Constant Width)

在示例中用来显示你逐字输入的文本，在正文中则用来显示代码。数据值用加引号的等宽字体 (monospace) 显示，引号不作为值的一部分。

倾斜等宽字体 (*Constant Width Oblique*)

用于必须替换为特定值的代码通名。有时也在示例中用来显示一个程序生成的输出。

加粗等宽字体 (**Constant Width Bold**)

有时用来表示要在命令行shell中逐字输入的文本。

加粗倾斜等宽字体 (***Constant Width Bold Oblique***)

用来表示需要与shell输入相区别的文字输出。

---

注4： Request Tracker即RT的创建者Best Practical公司对于一些重要的Perl项目免费提供了服务，包括Perl本身以及每一个CPAN版本。

我们提供了大量示例，它们大多是一个更大程序中的部分代码段。有些例子甚至是完整的程序，你应该能看出来，因为它们都以`#!`开头。几乎所有较长的程序最前面都有以下代码行：

```
#!/usr/bin/perl
```

还有一些示例需要在命令行上输入。我们用`%`表示通用的shell提示符：

```
% perl -e 'print "Hello, world.\n"'
Hello, world.
```

这种风格代表一个标准的Unix命令行，这里的单引号表示“最常用的引号”形式。其他系统上的引号和通配符约定可能有所不同。例如，如果需要对参数分组，而参数中包含空格或通配符时，MS-DOS和VMS系统上的很多命令行解释器需要双引号而不是单引号。

## 致谢

这里我们要广泛感谢为我们提意见的朋友、技术顾问以及审校人员，以弥补我们私底下对他们有过的种种抱怨：Abigail、Matthew Barnett、Piers Cawley、chromatic、Damian Conway、Dave Cross、Joaquin Ferrero、Jeremiah Foster、Jeff Haemer、Yuriy Malenkiy、Nuno Mendes、Steffen Müller、Enrique Nell、David Nicol、Florian Ragwitz、Allison Randal、Chris Roeder、Keith Thompson、Leon Timmermans、Nathan Torkington、Johan Vromans和Karl Williamson。如果书中还存在技术错误，那都是我们之过，与他们无关。

还要向O'Reilly出版社的整个制作团队特别表示感谢，感谢他们付出的巨大努力，克服不计其数的艰难险阻和严峻挑战，终于在这个后现代出版世界中让这本书成功付印出版。首先要特别感谢我们的制作编辑Holly Bauer，感谢她在我们交稿后付出极大的耐心，“吹毛求疵”地指出了数千个需要修改和增补的地方。感谢产品经理Dan Fauxsmith帮我们搜寻那些Unicode例子的罕见字体，并保证这本书的整个制作过程平稳进行。还要感谢制作指导Adam Witwer的亲力亲为，全力以赴攻克Antenna House格式化软件的种种问题，制作完成了这本书的制版稿。最后，我们要感谢出品人Laurie Petrycki，感谢他支持这个团队创造出作者们希望出版的书，还要感谢他鼓励这些作者写出人们想看的书。

## Safari®图书在线

Safari®图书在线 ([www.safaribooksonline.com](http://www.safaribooksonline.com)) 是一个应需而变的数字图书馆，可以供你方便快捷地在7500种技术和创意类参考书以及视频中找到需要的信息。

订阅之后，可以在这个图书馆在线阅读、查看所有图书及视频。你可以在手机和移动设备上阅读，在新书付印之前就能一睹为快，还可以查看正在编写的手稿并向作者直接反馈。



Safari®图书在线允许你复制粘贴示例代码、管理收藏文件夹、下载章节、给关键段落加书签、创建批注、打印页面，还提供了很多其他特性，可以大大节省你的时间。

O'Reilly公司已经将本书上传至Safari®图书在线服务。要获得本书及其他来自O'Reilly等出版社类似图书的电子版完全访问权限，请到<http://my.safaribooksonline.com>免费注册。

## 欢迎提出宝贵意见

我们已经尽最大努力对这本书中的内容做了全面测试和验证，不过你会发现有些内容可能有变化（另外我们也可能有错误和疏漏）。如果你发现有什么错误，或者对将来的版本有什么建议，请通过下面的地址告诉我们：

美国：

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）  
奥莱利技术咨询（北京）有限公司

我们为这本书建有一个网站，用于列出勘误以及其他与这本“骆驼书”有关的信息：

<http://shop.oreilly.com/product/9780596004927.do>

在这个网站上你还能找到本书中的所有示例代码，可以直接下载，而不用像我们这样一个字符一个字符地敲入。

如果想就本书发表评论或咨询技术问题，请发送Email至：

[bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)

关于我们的图书、会议、资源中心以及O'Reilly Network，请访问我们的网站：

<http://www.oreilly.com>

<http://www.oreilly.com.cn>

# 概述





# Perl概述

## Perl入门

我们认为Perl是一种易学易用的语言，希望能向你证明这一点。之所以说Perl很简单，这是因为如果你想表达什么，根本无须说太多。很多编程语言中，在写第一条可执行的代码语句之前，你必须声明要用到的类型、变量和子例程。当然，对于需要复杂数据结构的复杂问题，声明确实是个好主意。不过，对于很多简单的日常问题，你肯定更希望有一种更方便的编程语言，只需要简单地说：

```
print "Howdy, world!\n";
```

程序就能如你所愿。

Perl就是这样一种语言。实际上，这个例子也确实是一个完整的程序<sup>注1</sup>，如果把它输入到Perl解释器里，它就会在屏幕上打印出“Howdy, world!”（这个例子中的\n会在输出末尾产生一个换行符）。

就这么简单。说完你想说的话之后，你也不用再多说什么。与很多其他语言不同，Perl认为程序结束是退出程序的一种正常方式。如果你愿意，当然也可以显式地调用exit函数，同样的，还可以声明一些变量，甚至可以要求自己声明所有变量。不过，这些都由你来选择。利用Perl，不论你是否想提供定义，都能做好该做的事情。

Perl之所以易于使用还有很多其他原因，不过没有必要在这里全部列出来，因为这正是这本书后面要讨论的内容。俗话说，细节决定成败，不过Perl会努力把你从细节中解救出

---

注1： 或者也可以称为脚本、应用或可执行文件，甚至可以叫做小玩意，怎么都行。

来。在每一个层次上，Perl都会尽力帮助你直达目标，麻烦最少，享受最多。这也是为什么很多Perl程序员总是脸上带笑的原因。

这一章只是对Perl的一个概述，所以我们并不指望你能通过这一章从理性角度深刻认识Perl。另外我们也不追求完整性和逻辑性。这些是后面各章要做的。急脾气的人可以跳过这个概述直接进入第2章，用更短的时间获得最大的信息量。另一方面，如果你需要一个循序渐进的教程，那么向你推荐《Learning Perl》。不过可别把这本书弃之一边。

这一章是让你从另一个角度认识Perl，可以称之为关联性、艺术性、热情，或者甚至是松软性。为此，我们会向你展现Perl的各个视角，使你对Perl逐步建立一个清晰的认识，就像盲人摸象一样（当然，我们会做得更好，可能比那些瞎子强多了）。我们面对的是一个骆驼（见封面）。在Perl的这些视角中，衷心希望至少有一个角度能帮你摆脱困境。

## 自然语言和人工语言

语言最早是人类为自己发明的。但在计算机科学的演进过程中，人们有时会忘记这一点<sup>注2</sup>。不严格地讲，由于Perl碰巧是由一位语言学家设计的，因此被设计为能像自然语言一样顺畅使用。当然，这涉及很多方面，因为自然语言可以同时多个层次上有效工作。这里可以列举出这样一些语言学原则，不过语言设计中最重要一个原则是：容易的事情应当很容易实现，困难的事情也应当有可能完成（实际上，这是两个原则）。看起来似乎显而易见，不过很多计算机语言都没能完全满足这个原则，往往在某一方面有所欠缺。

自然语言在这两方面都做得很好，因为人们在一直不断地努力表达容易和困难的事情，所以自然语言得到了充分进化，可以很好地处理这两方面。最初设计Perl时就要求它应当能够演进，实际上它也确实做到了这一点。多年来，很多人都对Perl的演进做出了贡献。我们经常开玩笑说，委员会原本设计的是“马”，现在却变成了“骆驼”，不过再想想看，骆驼能很好地适应沙漠中的生活。骆驼已经进化为能够自给自足（另一方面，骆驼闻起来可不怎么样。Perl也一样）。我们选择骆驼作为Perl的吉祥物有很多奇怪的原因，不过这是原因之一，但与语言学没有太大关系。

如今，人们谈到“语言学”这个词时，很多人只会关注于一点，可能想到的是单词，也可能只想到了句子。不过单词和句子只是“拆分”语言的两种简单方法。可以划分为比较小的含义单位，或者将这些较小的单位合并为更大的含义单位。各个单位的含义很大程度上取决于这个单位所在的语法、语义和语用上下文。自然语言包含多种类型的单词，如名词、动词等。如果有人单独说到“狗”（dog），则你会认为这是一个名词，不过还可以用其他方式使用这个词。也就是说，如果上下文需要，则名词也可以作为动词、形容词或者副词。例如，在句子“If you dog a dog during the dog days of summer, you'll be a dog

---

注2：更确切地说，人们只是偶尔会记得这一点。



tired dogcatcher”<sup>注3</sup>（如果你在三伏天追赶一只狗，将会成为一个疲惫的捕狗人）中，“dog”一词就有多种不同词性。Perl也一样，在不同的上下文中会对单词做不同的处理。后面我们就会了解这一点。只要记住，Perl会努力理解你所说的，就像一个很好的听众。Perl会尽力恪守承诺。只要说出你的意思，Perl通常都能“理解”（当然，除非你在胡说八道，Perl解析器能更好地理解Perl，而不是英语或斯瓦希里语）。

再来考虑名词。名词可能指示一个特定的对象，也可能泛指一类对象（而不是指定某个具体对象）。大多数计算机语言都对这二者做了区分，只是我们将特定的对象称为值，而泛指的对象称为变量。值只存在于某个地方（可以是任何地方），不过变量在其生命期中会与一个或多个值关联。所以不论是谁来解释变量，都必须跟踪它的所有关联。这个解释器可能在你的大脑中，也可能在计算机中。

## 变量语法

变量就是用来保存某个东西的场所，它会有一个名字，这样以后需要回来寻找时，就能知道去哪里找你想要的那个特殊东西。就像在现实生活中一样，有各种不同的地方可以储存东西，有些很私密，有些则是公开的。有些地方是临时性的，另外一些地方可能是永久性的。计算机科学家们喜欢讨论变量的“作用域”，实际上就是这个意思。Perl提供了很多简便的方法来处理作用域问题，在后面适当的地方你就会学到，不过现在还不是时候（如果你很好奇，可以先查看第27章中的几个形容词：local、my、our和state，或者可以参阅第4章中的“作用域声明”一节）。

不过，要区分变量，更直接的方法是看变量中包含什么类型的数据。就像在英语中一样，Perl变量类型的主要区别在于单数和复数数据。字符串和数字是单数，而字符串或数字列表是复数（谈到面向对象编程时，你会发现一般对象从外部看是单数，而从内部看是复数，这就类似于一个班的学生）。我们把单数变量称为标量（scalar），复数变量称为数组（array）。由于字符串可以存储在标量变量中，可以把前面的第一个例子改写为稍长一些的版本（加了注释），如下所示：

```
my $phrase = "Howdy, world!\n";    # 创建变量
print $phrase;                     # 显示这个变量
```

这会告诉Perl，这个\$phrase是一个全新的变量（原来没有这样一个变量），所以不用去查找原来的变量。需要说明，我们不必非常特定地指出变量\$phrase是什么类型。\$字符会告诉Perl这个phrase是一个标量变量；也就是说，这是一个包含单数值的变量。与之相反，数组变量要以@字符开头[这样记可能会有帮助：\$是增加了艺术效果的“s”，表示“标量”（scalar），而@是艺术处理过的“a”，表示“数组”（array）]<sup>注4</sup>。

注3： 你可能已经烦透了这些废话。不过说实在的，我们只是希望你能理解Perl与典型的计算机语言有什么不同。

注4： 这些印记的故事远不是这么简单，我们会在第2章详细介绍。



Perl还有其他一些变量类型，名字很怪异，比如“散列”（hash）、“句柄”（handle）和“类型团”（typeglob）。与标量和数组类似，这些变量类型前面都有一些有趣的字符作为前缀，这通常称为印记（sigils）。为完整起见，表1-1列出了你可能遇到的所有印记。

表1-1：变量类型及用法

类型	印记	示例	用法
标量（Scalar）	\$	\$cents	单个值（数字或字符串）
数组（Array）	@	@large	值列表，以数字作为键
散列（Hash）	%	%interest	一组值，以字符串作为键
子例程（Subroutine）	&	&how	一个可调用的Perl代码块
类型团（Typeglob）	*	*struck	所有名为struck的东西

一些纯粹的语言学者将这些印记作为诟病Perl的一个原因。这有些草率。印记有很多好处，其中很重要的一点是：利用印记，变量可以方便地插补到字符串中，而无需额外的语法。Perl脚本也很容易阅读（当然是对那些花时间学习Perl的人而言），因为名词与动词可以很好地区分开。另外可以在Perl中增加新动词而不必担心破坏原来的脚本（我们说过，Perl最初就设计为一种可以进化的语言）。指定名词标记也很重要，英语和其他语言中已经有大量这样的先例，都需要有指示文法的名词标记。我们也是这样考虑的！

单数变量

从前面的例子可以看到，与很多其他计算机语言一样，可以用=操作符为标量赋一个新值。标量变量可以赋为任何形式的标量值：整数、浮点数、字符串，甚至可以赋为一些晦涩难懂的东西，比如其他变量或对象的引用。生成这些值来完成赋值有很多方法。

与UNIX<sup>注5</sup> shell中一样，可以使用不同的引号机制来建立不同类型的值。双引号(double quotes) 完成变量内插（variable interpolation）<sup>注6</sup>和反斜线内插（backslash interpolation，如将一个\n转换为换行符），而单引号会取消内插。反斜线（向左倾斜）会执行一个外部程序，并返回程序的输出，所以可以捕获为一个字符串，其中包含所有输出行。

```
my $answer = 42;           # 一个整数
my $pi = 3.14159265;       # 一个实数
```

注5：只要提到UNIX，我们指的是所有类UNIX的操作系统，包括BSD、Mac OS X、Linux、Solaris、AIX，当然还有UNIX。

注6：有时shell程序员也把这称为“变量替换”，不过Perl中“替换”一词还有其他含义，所以我们还是将“替换”留作他用。这里就称之为“内插”。我们按它的文本意义来使用（如“这段文字是诺斯替插入文字”），而不按是其数学含义（如“图中这个点是另外两点间的插值点”）。

<code>my \$avocados = 6.02e23;</code>	<code># 科学计数法</code>
<code>my \$pet = "Camel";</code>	<code># 字符串</code>
<code>my \$sign = "I love my \$pet";</code>	<code># 有内插的字符串</code>
<code>my \$cost = 'It costs \$100';</code>	<code># 没有内插的字符串</code>
<code>my \$thence = \$whence;</code>	<code># 另一个变量的值</code>
<code>my \$salsa = \$moles * \$avocados;</code>	<code># 一个表达式</code>
<code>my \$exit = system("vi \$file");</code>	<code># 一个命令的数值状态</code>
<code>my \$cwd = `pwd`;</code>	<code># 一个命令的字符串输出</code>

以上还没有涉及一些有趣的值，需要指出，标量还可以包含其他数据结构的引用，包括子例程和对象。

<code>my \$ary = \@myarray;</code>	<code># 一个命名数组的引用</code>
<code>my \$hsh = \%myhash;</code>	<code># 一个命名散列的引用</code>
<code>my \$sub = \&amp;mysub;</code>	<code># 一个命名子例程的引用</code>
<code>my \$ary = [1,2,3,4,5];</code>	<code># 一个未命名数组的引用</code>
<code>my \$hsh = {Na =&gt; 19, Cl =&gt; 35};</code>	<code># 一个未命名散列的引用</code>
<code>my \$sub = sub { print \$state };</code>	<code># 一个未命名子例程的引用</code>
<code>my \$fido = Camel-&gt;new("Amelia");</code>	<code># 一个对象的引用</code>

创建一个新的标量变量时，在为其赋值之前，它会自动用一个称为`undef`的值完成初始化，应该可以猜到，`undef`就表示“未定义”。取决于具体的上下文，这个未定义的值可能会解释为一个更为确定的`null`（空）值，如`"`或`0`。更一般地，根据你将如何使用这些变量，变量将自动解释为字符串、数字或者“`true`”和“`false`”值（通常称为布尔值）。应该记得人类语言中上下文语境是何等重要。在Perl中，各种操作符需要某些类型的单数值作为参数，所以我们称这些操作符在向那些参数“提供”或“供应”标量上下文。有时，我们还会更为特定，指出操作符为参数提供了一个数值上下文、字符串上下文或布尔上下文（后面我们还会谈到列表上下文，这与标量上下文完全不同）。Perl会自动将数据转换为当前上下文所需的合理形式。例如，假设有以下代码：

```
my $camels = "123";
print $camels + 1, "\n";
```

`$camels`的第一个赋值是一个字符串，不过它会转换为一个数字以便与1相加，然后再转换回字符串，从而打印出124。换行符表示为`"\n"`，它也在字符串上下文中，但由于它已经是一个字符串，所以不需要做任何转换。不过要注意，这里必须使用双引号，如果使用单引号（`'\n'`），则会得到一个两字符的字符串（包含一个反斜线和一个“`n`”），不论对谁而言，这都绝对不是一个换行符。

所以从某种意义上说，双引号和单引号也是一种指定上下文的方式。对于一个加引号的字符串，如何解释其内部内容取决于你使用哪一种引号〔后面我们还会看到另外一些文法与引号类似的操作符，不过会用某种特殊的方式使用字符串，如完成模式匹配或替换。这些都与双引号字符串类似。双引号（`doublequote`）上下文是Perl的“内插”上下文，很多与双引号类似的操作符可以提供这种内插上下文〕。



同样地，如果提供一个“解引用”上下文，引用才会表现为一个引用，否则只相当于一个简单的标量值。例如，有以下代码：

```
my $fido = Camel->new("Amelia");  
if (not $fido) { die "dead camel"; }  
$fido->saddle();
```

在这里我们创建了Camel对象的一个引用，把它放在一个新变量\$fido中。下一行中，将\$fido作为一个标量布尔变量进行测试，判断它是否为“true”，如果不为true，则抛出一个异常（也就是说，我们会抱怨），在这里意味着Camel->new构造函数没能创建一个合法的Camel对象。不过在最后一行上，我们又把\$fido看作是一个引用，要求它为\$fido中包含的对象（在这里就是Camel）查找saddle方法，所以Perl会查找Camel对象的saddle方法。有关的更多内容将在后面介绍。现在只要记住，在Perl中上下文很重要，因为Perl可以根据上下文知道你想要什么，而不必像很多其他计算机语言那样要求你明确地告诉它。

## 复数变量

有些变量包含逻辑上关联在一起的多个值。Perl有两类多值变量：数组和散列。在很多方面，它们都与标量类似。例如，新变量可以用my声明，另外它们也会自动初始化为一个空状态。不过，它们与标量有一个重要差别：为这些多值变量赋值时，会为赋值等式右边提供列表（list）上下文，而不是标量上下文。

数组和散列也各有不同。想按数字查找元素时，可以使用数组。想按名字来查找时，就要使用散列。这两个概念是相辅相成的——你经常会看到人们使用数组把月份（数字）转换为月份名（字符串），再用一个对应的散列将月份名（字符串）转换回月份（数字）（不过，散列并不仅限于保存数字，例如，还可以建立一个散列将月份名转换为生日幸运石名字）。

**数组。**数组（array）是一个有序的标量列表，按标量在列表中的位置来访问<sup>注7</sup>。这个列表可以包含数字、字符串或者二者同时包含（还可以包含子数组或子散列的引用）。要为一个数组赋一个列表值，只需要（用一对小括号）把这些值组织在一起：

```
my @home = ("couch", "chair", "table", "stove");
```

反之，如果在列表上下文中使用@home（如放在列表赋值等式的右边），就能取回原来放入的列表。可以由数组创建4个标量变量，如下所示：

```
my ($potato, $lift, $tennis, $pipe) = @home;
```

这称为列表赋值。从逻辑上讲，这些赋值是并行的，所以可以如下交换两个已有的变量：

```
($alpha,$omega) = ($omega,$alpha);
```

---

注7：也可以说是以此作为键或下标，或者称为索引或查找。随你选择。



与C中一样，数组从0开始计数，所以如果谈到数组中第1个到第4个元素，就要用下标0到3来访问<sup>注8</sup>。数组下标用中括号括起（[就像这样]），如果你想选择某个数组元素，要表示为\$home[n]，这里n就是你想要的元素下标（元素编号减1）。请看下面的例子。由于你处理的元素是一个标量，所以前面总要加上\$前缀。

如果希望一次为一个数组元素赋值，这也是可以的；数组元素会根据需要自动创建，所以可以把前面的赋值语句重写为：

```
my @home;  
$home[0] = "couch";  
$home[1] = "chair";  
$home[2] = "table";  
$home[3] = "stove";
```

可以看到，可以用my创建一个变量，而不必指定初始值（不需要对单个元素使用my，因为数组已经存在，它知道如何根据需要创建元素）。

由于数组是有序的，可以对它们做各种有用的操作，如堆栈操作push和pop。毕竟，堆栈就是一个有表头和表尾的有序列表，特别是有一个表尾。Perl把数组末尾当作栈顶（不过，大多数Perl程序员会把数组看作是水平的，栈顶在右边）。

**散列。**散列（hash）是一个无序的标量集合，利用与各标量关联的某个字符串值来访问<sup>注9</sup>。出于这个原因，散列通常也称为关联数组（associative arrays）。不过对于懒于敲键的人来说，这个名字太长了，所以我们决定起一个简短爽快的名字。选择“散列”还有一个原因，我们想以此强调它们是无序的（实际上，散列在内部使用了散列表查找，这也是散列操作如此快速的原因，而且不论在散列中放入多少个值，它也能一直保持它的快速操作）。不过，不能对散列执行push或pop操作，因为这是没有意义的。散列没有表头和表尾。但要知道，散列确实很强大也很有用。如果不能很好地理解和应用散列，就不能真正了解Perl。图1-1显示了有序的数组元素和有序（但已命名）的散列元素。

由于不能由元素位置自动指示散列的键，所以在填充散列时必须同时提供键和值。仍然可以像普通数组一样用一个列表为散列赋值，不过列表中的每一对元素将分别解释为一个键和一个值。由于我们要处理元素对，散列使用%印记来标记散列名（如果仔细观察%字符，可以看到这就像有一个键和一个值，二者之间有一个斜线。这样看可能有助于记忆）。

假设你想把简写的星期名转换为相应的全名。可以写出以下列表赋值：

---

注8： 如果你觉得这很奇怪，可以把下标看作是偏移量；也就是说，数组中在它之前有多少个元素。很显然，第1个元素前面没有任何元素，所以它的偏移量就是0。计算机就是这么考虑的（在我们看来）。

注9： 也可以说是以此作为键或下标，或者称为索引或查找。随你选择。

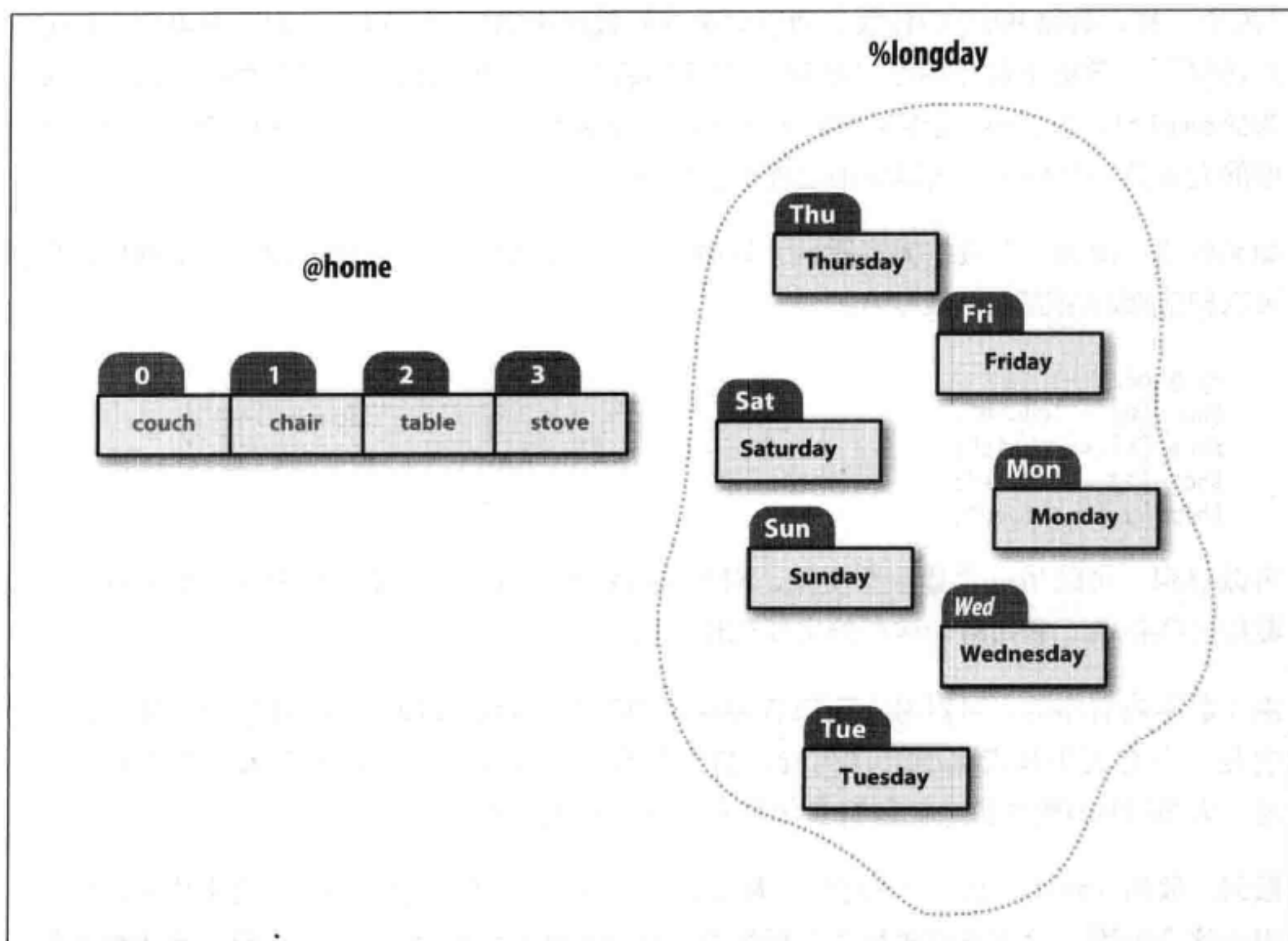


图1-1：数组和散列

```
my %longday = ("Sun", "Sunday", "Mon", "Monday", "Tue", "Tuesday",
               "Wed", "Wednesday", "Thu", "Thursday", "Fri",
               "Friday", "Sat", "Saturday");
```

不过这读起来很费劲，所以Perl提供了`=>`（等号和大于号）序列来取代逗号。使用这种表示法（再加上一些有创意的格式化处理），就能更容易地看出哪些字符串是键，而哪些字符串是与键关联的值。

```
my %longday = (
    "Sun" => "Sunday",
    "Mon" => "Monday",
    "Tue" => "Tuesday",
    "Wed" => "Wednesday",
    "Thu" => "Thursday",
    "Fri" => "Friday",
    "Sat" => "Saturday",
);
```

如上所述，可以用列表为散列赋值，不仅如此，如果在列表上下文中提到散列，它会把这个散列转换回一个键/值对列表（可能顺序有些奇怪）。有时这会很有用。大多数情况下，人们只是使用`key`函数（这个函数名很恰当）从列表中抽取键。这个键列表也是无序

的，不过如果需要，只需使用`sort`函数（同样函数名很恰当）就能很容易地对它进行排序。然后，可以使用有序的键按你希望的顺序取出相应的值。

由于散列是一种特别的数组，选择单个散列元素时，要用大括号（这些大括号也称为“花括号”）将键括起来。例如，如果想在上面的散列中找出与`Wed`关联的值，就要使用`$longday{"Wed"}`。重申一次，你处理的是一个标量值，所以最前面要用`$`而不是`%`，`%`表示的是整个散列。

从语言学角度讲，散列中体现的关系是一种所有格关系，类似英语中的单词“`of`”，或者类似于英语的“`'s`”（在中文中，类似于“…的”）。如果说“Adam的妻子是Eve”（The wife of Adam is Eve），可以写作：

```
my %wife;
$wife{"Adam"} = "Eve";
```

## 复杂数据结构

数组和散列是简单、可爱的扁平数据结构。遗憾的是，尽管我们希望能简化再简化，但理想和现实总是有差距。有时你还需要构建不那么简单、不那么可爱、不那么扁平的数据结构。利用Perl能轻松地做到这一点，可以假装那些复杂值只是简单值。换句话说，Perl允许你处理简单的标量引用，可能这些引用恰好指示复杂的数组和散列。在自然语言中我们一直是这样做的，总是使用简单的单数名词来表示复杂的事物，比如用“政府”这个词表示一个复杂而费解的实体。这样的例子还有很多。

延续前面的例子，假设我们不想再讨论Adam的妻子，现在来讨论Jacob的妻子。Jacob有4位妻子（你自己可别这么干）。要在Perl中表示这一点，你会发现这种情况很奇怪，我们想把Jacob的4位妻子当成一个妻子（同样建议你不要这么做）。你可能以为可以写这样一行语句来表示：

```
$wife{"Jacob"} = ("Leah", "Rachel", "Bilhah", "Zilpah");           # 错误
```

不过，这样并不能如你所愿，因为在Perl中，小括号和逗号还没有强大到可以把列表转换为一个标量（小括号只用于语法分组，逗号用于语法分隔）。实际上，你要明确地告诉Perl希望把一个列表假装成一个标量。中括号有能力做到这一点：

```
$wife{"Jacob"} = ["Leah", "Rachel", "Bilhah", "Zilpah"];           # 正确
```

这个语句会创建一个未命名的数组，将这个数组的一个引用放在散列元素`$wife{"Jacob"}`中。这样我们就有了一个命名散列，其中包含一个未命名的数组。Perl就是以这种方式来处理多维数组和嵌套数据结构的。与普通数组和散列一样，也可以对单个元素赋值，如下所示：

```
$wife{"Jacob"}[0] = "Leah";
$wife{"Jacob"}[1] = "Rachel";
```



```
$wife{"Jacob"}[2] = "Bilhah";
$wife{"Jacob"}[3] = "Zilpah";
```

可以看到这像是一个多维数组，包含一个字符串下标和一个数值下标。要看一个树状结构，比如嵌套数据结构，下面假设我们希望不仅列出Jacob的多位妻子，还要列出每位妻子的所有儿子。这种情况下，我们希望把散列当作一个标量。可以用大括号来实现（每个散列值中，将如前所示使用中括号表示数组。不过现在是在一个嵌套的散列中建立数组）。

```
my %kids_of_wife;
$kids_of_wife{"Jacob"} = {
    "Leah" => ["Reuben", "Simeon", "Levi", "Judah", "Issachar", "Zebulun"],
    "Rachel" => ["Joseph", "Benjamin"],
    "Bilhah" => ["Dan", "Naphtali"],
    "Zilpah" => ["Gad", "Asher"],
};
```

这就等价于：

```
my %kids_of_wife;
$kids_of_wife{"Jacob"}{"Leah"}[0] = "Reuben";
$kids_of_wife{"Jacob"}{"Leah"}[1] = "Simeon";
$kids_of_wife{"Jacob"}{"Leah"}[2] = "Levi";
$kids_of_wife{"Jacob"}{"Leah"}[3] = "Judah";
$kids_of_wife{"Jacob"}{"Leah"}[4] = "Issachar";
$kids_of_wife{"Jacob"}{"Leah"}[5] = "Zebulun";
$kids_of_wife{"Jacob"}{"Rachel"}[0] = "Joseph";
$kids_of_wife{"Jacob"}{"Rachel"}[1] = "Benjamin";
$kids_of_wife{"Jacob"}{"Bilhah"}[0] = "Dan";
$kids_of_wife{"Jacob"}{"Bilhah"}[1] = "Naphtali";
$kids_of_wife{"Jacob"}{"Zilpah"}[0] = "Gad";
$kids_of_wife{"Jacob"}{"Zilpah"}[1] = "Asher";
```

从上面可以看到，为一个嵌套数据结构增加一层，就像是为一个多维数组再增加一维。Perl中这两种理解都可以，不过内部表示是一样的。

这里的重点是，Perl允许你把一个复杂的数据结构当成一个简单的标量。正是基于这种简单的封装，构建了Perl的整个面向对象结构。之前调用Camel构造函数时（如下所示）：

```
my $fido = Camel->new("Amelia");
```

我们创建了一个Camel对象，由标量\$fido表示。不过这个Camel的内部更为复杂。作为优秀的面向对象程序员，我们并不关心Camels的内部细节（除非我们刚好要实现Camel类的方法）。不过，一般来讲，像Camel这样的对象通常由一个散列组成，其中包含特定Camel对象的属性，如骆驼名（在这里，骆驼名是“Amelia”而不是“fido”）和驼峰数（我们没有指定具体的驼峰数，不过可能默认为1，请参考封面）。

## 简单数据结构

读完上一节如果还没有觉得头晕脑胀，你可真不简单。人们通常不喜欢处理复杂的数据结构，不论是官方还是非官方。所以，在我们的自然语言中，有很多办法来隐匿复杂性。其中一些方法可以归入“主题化”（topicalization）一类，这是一个很有意思的语言学概念，表示在与某人谈论时，对你们将要谈论的主题（也包括你们不打算谈论的主题）达成共识。这会在语言中的多个层次出现。在高层次上，我们自己会划分为多个子文化，分别关注不同的子主题，并建立子语言来讨论这些子主题。医生办公室中的语言（如“不可溶解窒息物”）就与巧克力工厂里的语言（如“美味持久糖球”）大不相同。从一种语言变到另一种语言时，我们大多会自动切换上下文。

在会话层次，这种上下文切换必须更为明确，所以语言为我们提供了很多方式来表达我们要讨论的主题。书有书名，章节有标题。在句子中我们可能会加上这样的语句“根据最新调查”或者“对于所有X”来引出主题。不过在谈话中，我们通常都会开门见山地直接问，比如“你知道喉咙后面的东西是什么？”

Perl也提供了多种主题化方式。其中一个重要的主题化方法是package声明。假设你想在Perl中谈论骆驼（Camel）。可能会用以下声明开始你的Camel模块：

```
package Camel;
```

这有很多重要的影响。其中一个结果是，Perl会认为从这里开始所有全局动词或名字都是有关于骆驼的。为此它会自动为所有全局名<sup>注10</sup>增加一个模块名“Camel::”作为前缀。所以如果有以下代码：

```
package Camel;
our $fido = &fetch();
```

\$fido的真实名字则是\$Camel::fido（&fetch的真实名字是&Camel::fetch，不过我们还没有谈到动词）。这意味着如果其他模块有以下代码：

```
package Dog;
our $fido = &fetch();
```

Perl不会搞混，因为这个\$fido的真实名字是\$Dog::fido而不是\$Camel::fido。计算机科学家称之为“包（package）建立了一个命名空间（namespace）”。可以根据需要建立多个命名空间，不过一次只能在一个命名空间中，此时可以假装其他命名空间都不存在。就是通过这种方式，命名空间可以为你简化现实。这种简化以“假装”为前提（当然，还可以更为简化，方法也是一样的，这正是这一章要讲的）。

---

注10：可以使用our声明全局变量，它与my很类似，不过会告诉人们这是一个共享变量。my变量并不共享，在当前代码块以外是看不到的。如果不确定使用my还是our，最好用my而不是our，因为不必要的全局变量只会带来麻烦和困惑。



保证名词的含义明确固然重要，保证动词的含义明确也同样很重要。Camel和Dog命名空间中的`&Camel::fetch`与`&Dog::fetch`不会混淆，这很好，不过包真正的好处在于能够对动词分类，让其他包也能使用这些动词。如果有下面的代码：

```
my $fido = Camel->new( "Amelia" );
```

实际上我们在调用Camel包中的`&new`动词，其全名为`&Camel::new`。如果执行下面的代码：

```
$fido->saddle();
```

则是在调用`&Camel::saddle`例程，因为`$fido`记得它指向一个Camel。这就是面向对象编程。

如果声明`package Camel`，会开始一个新的包。不过有时你只是想借用一个已有包中的名词和动词。这在Perl中是允许的，可以利用一个`use`声明来实现，这样不仅可以从另一个包借用动词，还可以检查确认从磁盘载入你指定的包。实际上，在执行下面这个语句之前，

```
my $fido = Camel->new( "Amelia" );
```

必须声明：

```
use Camel;
```

否则，Perl不知道Camel是什么。

有意思的是，假设你能让别人帮你写Camel模块，那么你自己并不需要知道Camel是什么。当然最好是有人已经为你写了Camel模块。尽管有争议，不过有人说Perl最强大之处不是Perl本身，而是CPAN（Comprehensive Perl Archive Network，Perl综合典藏网，见第19章），其中包含大量模块，可以完成很多不同的任务，你甚至无须了解它们是怎么做的。只需要下载你想要的模块，然后声明：

```
use Some::Cool::Module;
```

你就能针对所讨论的主题采用适当的方式使用这个模块中的动词了。

所以，与自然语言中的主题化类似，Perl中的主题化能够把你要使用的语言“归整”到特定作用域中。实际上，有些内置模块并没有引入任何动词，而只是以多种有用的方式归整Perl语言。我们把这些特殊的模块称为`pragmas`（参见第29章）。例如，我们经常看到人们使用`pragma strict`，如下：

```
use strict;
```

`strict`模块所做的只是强化一些规则，要求你对可能需要Perl猜测的某些方面做出更明确的说明，如希望如何指定变量的作用域<sup>注11</sup>。如果要完成规模很大的项目，显式声明会很

注11：更确切地讲，`use strict`要求在变量声明中使用`my`、`state`或`our`；否则它会假设未声明的变量为包变量，这会在以后带来麻烦。另外，有些构造多年来已经证实很容易导致错误，使用`use strict`后将不允许使用这些构造。



有帮助。默认地，Perl更适用于小项目，不过利用strict pragma，Perl也同样适用于更要求可维护性的大型项目。由于可以在任何时候增加strict pragma，而小项目很可能会演变为大项目（即使你当初没有想到），因此Perl也能助你一臂之力。这种情况很常见。

随着Perl的进化，Perl社区也在演进，其中一个变化是，对于默认情况下Perl的行为，社区的看法已经发生转变（这与原来要求Perl坚持一贯表现的态度有所冲突）。例如，大多数Perl程序员现在都认为程序最前面都应当加上“use strict”。随着时间推移，我们可能会累积很多这种“文化要求的”用于语言归整的pragma。另一个内置pragma是Perl的版本号，这是一种“元pragma”，告诉Perl可以在各个方面表现得像一种更现代的语言：

```
use v5.14;
```

这个特定声明会打开多个pragma（包括“use strict”<sup>注12</sup>）。还会启用一些新特性，如动词say，它会为你增加一个换行符（与print不同）。所以我们可以把第一个例子改写为：

```
use v5.14;
say "Howdy, world!";
```

这本书中的例子都采用Perl的v5.14版本；为你展示完整的程序时，我们会尽量记得加上use v5.14，不过如果只是提供一些代码段，希望你自己加上这个声明（如果你没有最新版本的Perl，可能无法运行我们给出的一些例子。比如，你要把say改回为print，还要加上一个换行符，不过最好还是升级你的Perl版本。要想正常执行say，起码需要use v5.10）。

## 动词

与典型的命令式计算机语言一样，Perl中的很多动词都是命令：它们会告诉Perl解释器做某个动作。另一方面，与自然语言类似，取决于上下文，Perl动词往往有多种不同的含义。如果一个语句以动词开头，这通常是纯命令语句，执行它完全是为了达到其副作用[我们有时称这些动词为过程（procedures），特别是用户自定义的动词]。一个常见的内置命令是say命令（实际上你确实已经见过了）：

```
say "Adam's wife is $wife{'Adam'}.";
```

它的副作用是生成所需的输出：

```
Adam's wife is Eve.
```

不过除了命令语气外，还有其他“语气”。有些动词用来问问题，这在条件语句（如if语句）中很有用。还有一些动词将其输入参数转换为返回值，就像菜谱会告诉你如何把原材

---

注12：这种隐式“约束”特性是v5.12新增的。参见第29章中的feature pragma。

料加工成一道可口（希望如此）的菜肴。我们习惯把这些动词称为函数（functions），这也是为了迎合数学家的习惯[他们似乎不知道英语中“functional”（功能）一词的一般含义]。

可以举一个内置函数的例子，如指数函数：

```
my $e = exp(1); # 2.718281828459左右
```

不过Perl并没有严格区分过程和函数。你会看到这些词总在交替使用。动词有时也称为操作符（内置动词）或子例程（用户自定义的动词）<sup>注13</sup>。不过随你怎么称呼，它们都返回一个值，这可能是一个有意义的值，也可能不是，你可以选择将其忽略（或者不忽略）。

随着后面内容的展开，我们还会看到其他一些例子，可以从中了解Perl与自然语言的相似之处。不过也可以从其他角度考察Perl。我们已经悄悄介绍了数学语言中的一些概念，如下标、加法和指数函数。不仅如此，Perl还是一种控制语言、黏合语言、原型语言、文本处理语言、表处理语言以及面向对象语言等。

不过，Perl同时也是一个普通而古老的计算机语言。下面我们将从这个角度来介绍。

## 一个平均分例子

假设你在为一个班级讲授Perl语言，想确定如何给出学生的成绩。你已经有班上每一个学生的一组考试分数，顺序是随机的。你需要一个合并的列表，包含每个学生的所有成绩以及他们的平均分。你有这样一个文本文件（假设名为grades）：

```
Noël 25
Ben 76
Clementine 49
Norm 66
Chris 92
Doug 42
Carol 25
Ben 12
Clementine 0
Norm 66
...
```

可以使用以下脚本把他们的分数收集在一起，确定每个学生的平均分，然后按字母顺序打

---

注13：历史上，Perl要求在所有用户自定义子例程调用上增加一个&字符（见前面的`$fido = &fetch();`）。不过在Perl v5中，这个&字符是可选的，所以调用用户自定义动词时可以与调用内置动词一样使用同样的语法（`$fido = fetch();`）。谈到例程名时我们仍会使用&符号，如得到例程的一个引用时（`$fetcher = \&fetch;`）。从语言学角度讲，可以把加&号的&fetch看作是一个不定式“to fetch”或类似形式“do fetch”。不过如果可以直接说“fetch”，我们很少会说“do fetch”。这正是V5中不再强制要求增加&符号的真正原因。

印出来。这个程序简单地假设班上没有重名的学生（比如，不会有两个Carol）。也就是说，如果出现Carol的第二条记录，则程序会认为这只是第一个Carol的另一个分数（不会与第一个Noël混淆）。

顺便说一句，这里的行号并不是程序的一部分，不过其他方面都与BASIC很类似。

```
1  #!/usr/bin/perl
2  use v5.14;
3
4  open(GRADES, "<:utf8", "grades") || die "Can't open grades: $!\n";
5  binmode(STDOUT, ':utf8');
6
7  my %grades;
8  while (my $line = <GRADES>) {
9      my ($student, $grade) = split(" ", $line);
10     $grades{$student} .= $grade . " ";
11 }
12
13 for my $student (sort keys %grades) {
14     my $scores = 0;
15     my $total = 0;
16     my @grades = split(" ", $grades{$student});
17     for my $grade (@grades) {
18         $total += $grade;
19         $scores++;
20     }
21     my $average = $total / $scores;
22     print "$student: $grades{$student}\tAverage: $average\n";
23 }
```

离开这个例子之前，需要指出这里展示了目前为止介绍过的很多内容，以及我们将要解释的很多内容。不过，如果你将目光稍稍放远一点，可能会发现一些有趣的模式。可以大胆猜猜看它们会做什么，接下来就会告诉你猜得对不对。

应该让你试着运行这个程序，不过你可能还不知道该怎样运行。

## 如何运行

哈，现在你可能很想知道该怎么运行一个Perl程序。如果简单地回答，只需要把程序输入到Perl语言解释器中，这个解释器程序恰好就名为Perl。如果更详细地回答，开头语往往是：条条大路通罗马（There's More Than One Way To Do It）<sup>注14</sup>。

第一种调用Perl的方法（这也是几乎在所有操作系统上都可用的一种方法）就是直接从命

---

注14：这也是Perl的宣传口号，你可能早已经听腻了，除非你是本地专家（如果是这样，你肯定已经说腻了）。有时这个口号也简写为TMTOWTDI，拼作“tim-toady”。不过随你怎样拼都可以。毕竟，TMTOWTDI。



令行显式调用perl<sup>注15</sup>。如果你要做的事情很简单，可以使用-e开关（下面例子中的%表示一个标准shell提示符，不需要输入）。在UNIX上，可以输入：

```
% perl -e 'print "Hello, world!\n";'
```

在其他操作系统上，可能必须对引号做些调整。不过基本原则都是一样的：你要把Perl需要知道的所有一切都“塞”到80列左右的字符中<sup>注16</sup>。

对于比较长的脚本，可以使用你喜欢的文本编辑器（或者任何其他文本编辑器）把所有命令放在一个文件中，假设将这个脚本命名为gradation（不要与graduation混淆），可以执行：

```
% perl gradation
```

这里还是在显式地调用Perl解释器，不过至少不用每次都在命令行中输入所有代码。另外也不必为了让shell满意而反复调整引号。

要调用脚本，最方便的方法就是直接输入脚本名（或者点击这个脚本），让操作系统帮你找到合适的解释器。有些系统可能提供了一些方法，可以将不同文件扩展名或目录与某个特定的应用相关联。在这些系统上，你要想办法将Perl脚本与perl解释器关联起来。在支持#!“shebang”符号的UNIX系统上（如今大多数UNIX系统都会支持），可以让脚本的第一行代码具有“魔力”，使操作系统知道该运行哪个程序。参考前面例子中的第1行代码，可以在你的程序中放入类似的一行代码：

```
#!/usr/bin/perl
```

（如果perl v5.14不在/usr/bin中，则要相应地修改#!代码行<sup>注17</sup>。）然后只需执行：

```
% gradation
```

当然，这并不能正常工作，因为你还没有确保脚本是可执行文件（参见chmod(1)手册页），而且它必须要在你的PATH中。如果脚本不在你的PATH中，就必须提供脚本的完整文件名，以便操作系统知道该如何找到你的脚本。比如可以像这样：

```
% /home/sharon/bin/gradation
```

---

注15：假设你的操作系统提供了一个命令行界面。如果没有，请升级系统。

注16：这一类脚本通常称为“单行脚本”。如果你与其他Perl程序员交流过，会发现有些人非常喜欢创建这种复杂难懂的单行脚本。就是因为这些“家伙”，Perl有时被污蔑成一种“只写”语言。

注17：如果你的/usr/bin/perl是一个老版本，可以编译一个新版本，只要相应地调整#!代码行来指向这个解释器，将它放在任何位置都可以（如/usr/local/bin）。

最后一点，如果你很不幸，还在用老版本的UNIX系统（不支持“有魔法的”`#!`行），或者如果你的解释器的路径超过了32个字符（很多系统上都有这样一个内置限制），也可以另谋出路，如下所示：

```
#!/bin/sh -- # perl, 停止循环
eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
if 0;
```

有些操作系统上的默认命令解释器可能是`/bin/csh`、`DCL`、`COMMAND.COM`或者其他解释器，要处理这些解释器，则要求相应地修改以上代码。具体情况请咨询本地专家。

在本书中，我们将用`#!/usr/bin/perl`表示所有这些记法，不过你要知道我们实际的意思。

提示：编写一个测试脚本时，不要将这个脚本命名为`test`。UNIX系统有一个内置的`test`命令，尽管你想执行名为`test`的脚本，但系统可能会执行这个`test`命令，而不是你的脚本。可以把脚本名换作`try`。

你已经知道如何运行你自己的Perl程序（不要与perl程序混淆），下面再来看我们的例子。

## 文件句柄

除非是用人工智能为自我中心的哲学家建模，否则你的程序肯定都需要某种方式与外部世界沟通。在平均分例子的第4行到第8行中，可以看到`GRADES`，这是另一种Perl数据类型的例子，即文件句柄（filehandle）。文件句柄就是为文件、设备、套接字或管道指定的一个名字，可以帮你记住你所指的是哪一个文件或设备，另外可以隐藏缓冲等细节问题的部分复杂性（在内部，文件句柄就类似于C++等语言中的流，或者BASIC语言中的I/O通道）。

利用文件句柄，可以更容易地从很多不同位置得到输入，以及向不同位置发送输出。Perl之所以成为一种优秀的黏合语言，部分原因就在于它能一次与多个文件和进程通信。Perl能够为不同的外部对象提供明确的符号名，当然这只是成为一种优秀黏合语言的部分原因。<sup>注18</sup>

可以创建一个文件句柄，使用`open`将它与一个文件关联。`open`函数至少有两个参数：文件句柄以及希望与它关联的文件名。Perl还提供了一些预定义（而且预打开）的文件句柄。

---

注18：还有一些其他原因共同促使Perl成为一个优秀的粘合语言，这包括：它能处理非ASCII数据；它是可嵌入的，可以通过扩展模块在其中嵌入其他特性；它很简洁，而且很容易“联网”；它具有环境感知性（暂且这么说）；可以用多种不同方法调用（前面已经看到）。不过最重要的是，Perl语言本身的建构不那么严格，总有办法解决你的问题。这又回到了TMTOWTDI的话题。



STDIN是程序的正常输入通道，STDOUT是程序的正常输出通道。STDERR是另外一个输出通道，允许程序在将输入转换为（或试图转换为）输出时在一旁发出一些警告<sup>注19</sup>。在程序的第4行和第5行中，我们还告诉这个新的GRADES文件句柄和已有的STDOUT文件句柄假设文本采用UTF-8编码，这是一种常用的Unicode文本表示。

由于可以使用open函数为不同用途（输入、输出、管道）创建文件句柄，所以需要指定你希望做什么。如果使用命令行，可以直接为文件名增加一些特定的字符：

```
open(SESAME, "filename" )           # 读取现有文件
open(SESAME, "< filename")          # (同上，不过这里明确指定)
open(SESAME, "> filename")          # 创建文件并写入内容
open(SESAME, ">> filename")         # 追加到现有文件
open(SESAME, "| output-pipe-command") # 建立一个输出过滤器
open(SESAME, "input-pipe-command |")  # 建立一个输入过滤器
```

不过，推荐使用open的3参数形式，它允许在文件名之外的一个参数中指定打开模式。如果你处理的文件名并不是直接量，其中可能已经包含类似打开模式或空白符等字符，这种3参数形式会很有用。

```
open(SESAME, "<", $somefile)         # 读取现有文件
open(SESAME, ">", $somefile)         # 创建文件并写入内容
open(SESAME, ">>", $somefile)        # 追加到现有文件
open(SESAME, "|-", "output-pipe-command") # 建立一个输出过滤器
open(SESAME, "-|", "input-pipe-command")  # 建立一个输入过滤器
```

在我们的程序中，这种形式的open还允许指定文件的字符编码：

```
open(SESAME, "< :encoding(UTF-8)", $somefile)
open(SESAME, "> :crlf", $somefile)
open(SESAME, ">> :encoding(MacRoman)", $somefile)
```

可以看到，文件句柄可以选择任意的名字。一旦打开，文件句柄SESAME就可以用来访问文件或管道，直到将它显式关闭（可以想见，要用close(SESAME)来关闭），或者直到通过对同一个文件句柄再执行一次open将它关联到另一个文件。打开一个已经打开的文件句柄，这会隐式地关闭第一个文件，从而无法再通过这个文件句柄访问该文件，而打开另外一个文件。一定要仔细确认这确实是你想要的。有时这种情况会意外发生，比如你可能要执行open(\$handle,\$file)，而\$handle恰好包含一个常量字符串。务必将\$handle设置为某个独特的名字，否则你会用同一个文件句柄打开一个新文件。

一种更好的想法是保留\$handle仍为未定义，由Perl为你填充句柄。如果你不想再自己为文

---

注19：这些文件句柄通常都与终端关联，所以可以输入你的程序，并看到它的输出。不过文件句柄也可能关联到文件（等设备）。Perl会提供这些预定义的句柄，因为你的操作系统已经以某种方式提供了这些句柄。在Unix中，进程从其父进程（通常是shell）继承了标准输入、输出和错误通道。shell的职责之一就是建立这些I/O流，使子进程无须为此操心。



件句柄选择名字，这会很方便：倘若向open传入一个未定义的变量（如my creates），Perl会为你选择文件句柄，并自动填充：

```
open(my $handle, "< :crlf :encoding(cp1252)", $somefile)
|| die "can't open $somefile: $!";
```

如果open成功执行，现在\$handle变量就是已定义的，需要一个文件句柄时就可以使用这个变量。

一旦打开一个文件句柄来完成输入，就可以使用读行操作符<>读取一行。这也称为尖角操作符，因为它由尖括号组成。要从一个文件句柄读取一行，需要将这个文件句柄包围在尖角操作符中（<SESAME> 是一个直接量句柄，<\$handle>是一个间接句柄）。空尖角操作符<>将从命令行上指定的所有文件读取一行，如果没有指定任何参数，则从STDIN读取一行（这是很多过滤器程序的标准行为）。下面的例子使用了STDIN文件句柄读取用户提供一个答案，如下所示：

```
print STDOUT "Enter a number: ";      # 请求输入一个数
$number = <STDIN>;                    # 输入数字
say STDOUT "The number is $number.";  # 打印这个数字
```

有没有发现我们漏了什么？这些print和say语句中的STDOUT做什么用？嗯，这只是使用输出文件句柄的多种方式之一。在命令和参数表之间可以提供一个文件句柄，如果提供了文件句柄，它会指出输出到哪里。在这里，这个文件句柄是冗余的，因为不管怎样都会输出到STDOUT。就像STDIN是默认的输入通道一样，STDOUT是默认的输出通道。（在平均分例子的第22行中，我们省略了STDOUT，以免你糊涂）。

如果尝试前面这个例子，你会注意到会有额外的一个空行。这是因为，读行操作不会自动删除你的输入行中的换行符（例如，你的输入会是"9\n"）。有时可能确实想删除这个换行符，为此Perl提供了chop和chomp函数。chop会不加区别地删除（并返回）字符串的最后一个字符，而chomp只删除结束标记（通常是"\n"），并返回删除的字符数。以下是输入一行时惯用的做法，你会经常看到这种做法：

```
chomp($number = <STDIN>);      # 输入一个数字，然后删除换行符
```

这等价于下面的做法：

```
$number = <STDIN>;              # 输入一个数字
chomp($number);                 # 删除末尾的换行符
```

还有最后一点需要说明，我们把变量命名为\$number，但这并不意味着它就是一个数字。这可以是任何字符串。如果你想把这个字符串当作一个数字来处理，则Perl只关心是否有一个数字，这里就要谈到操作符，这也是下面我们要讨论的。

# 操作符

前面已经提到过，Perl还是一个数学语言。这在很多层次上都有体现，从底层的位逻辑操作符，上至数字和集合处理，再到更大的谓词和不同类型的抽象。我们都在学校里学过数学，都知道数学家总喜欢一些奇怪的符号。更糟糕的是，计算机科学家们又为这些奇怪的符号建立了他们自己的版本。Perl也有很多这样的奇怪符号，不过别担心，因为其中大多都是从C、Fortran、*sed*(1)或*awk*(1)直接借用的，所以至少对使用这些语言的人来说会很熟悉。

对其他人来说，值得安慰的是，通过学习Perl中的这些奇怪符号，这可以成为学习所有其他奇怪语言的一个良好开端。

Perl的内置操作符可以按操作数的个数分为一元操作符、二元操作符和三元操作符。也可以划分为前缀操作符（放在操作数前面）和中缀操作符（放在操作数中间）。还可以按所处理的对象类型来划分，如数字操作符、字符串操作符或文件操作符。后面会给出操作符表列出所有操作符，不过先来看一些简单常用的操作符。

## 一些二元算术操作符

算术操作符与我们在学校里学到的完全一样。它们会对数字执行某种数学运算，见表1-2。

表1-2：算术操作符表

示例	操作符名	结果
<code>\$a + \$b</code>	加法	<code>\$a</code> 和 <code>\$b</code> 之和
<code>\$a * \$b</code>	乘法	<code>\$a</code> 和 <code>\$b</code> 之积
<code>\$a % \$b</code>	求余	<code>\$a</code> 除以 <code>\$b</code> 的余数
<code>\$a ** \$b</code>	求幂	<code>\$a</code> 的 <code>\$b</code> 次幂

没错，我们没有列出减法和除法，相信你肯定知道它们如何工作。可以试试看与你所想的是否一样（也可以提前看看第3章）。算术操作符的执行顺序与数学老师所教的完全一致（先求幂，再是乘法，然后是加法）。可以用括号来改变运算顺序。

## 字符串操作符

字符串也有一个“加法”可以实现字符串的连接（也就是将字符串首尾相接）。很多语言中这个操作符总与完成数值相加的操作符混淆，Perl有所不同，它定义了一个单独的操作符(.)来完成字符串连接：



```
$a = 123;
$b = 456;
say $a + $b;      # 打印 579
say $a . $b;      # 打印 123456
```

字符串还有一个“乘法”操作符，这称为重复（repeat）操作符。类似地，Perl也用一个单独的操作符(x)表示字符串的重复，与数值乘法相区别：

```
$a = 123;
$b = 3;
say $a * $b;      # 打印 369
say $a x $b;      # 打印 123123123
```

与相应的数值操作符一样，这些字符串操作符优先级很高。重复操作符稍有些不同，它取一个字符串作为左参数，右参数却是一个数字。还要注意Perl会自动将数字转换为字符串。可以把以上的这些直接量数字用引号引起来，这样也会得到同样的输出。不过，在内部它已经从另一个方向完成了转换（即从字符串转换为数字）。

还有几个问题需要考虑。双引号字符串中的内插也会完成字符串连接。另外打印一个值列表时，实际上就是在连接字符串。所以，下面3个语句将得到相同的输出：

```
say $a . " is equal to " . $b . ".";  # 点操作符
say $a, " is equal to ", $b, ".";     # 列表
say "$a is equal to $b.";             # 内插
```

特定情况下究竟使用哪一种方式完全由你决定（不过，在我们看来，内插通常是最可读的）。

x操作符乍看起来似乎没有太大意义，不过有时它会非常有用，特别是类似下面这种情况：

```
say "-" x $scrwid;
```

这会在你的屏幕上画一条线（这里假设\$scrwid包含屏幕宽度，而不是标识符）。

## 赋值操作符

尽管简单的赋值操作符(=)不能完全算是一个数学操作符，不过我们已经大量使用了这个操作符。要记住=表示“设置为”而不是“等于”（另外还有一个表示相等的数学操作符==，它的含义才是“等于”。如果现在开始仔细考虑二者的区别，就能免去日后的很多麻烦。==操作符就像是一个返回布尔值的函数，而=更像是一个过程，执行这个过程的目的是为了得到其副作用，即修改一个变量）。

与前面介绍的操作符类似，赋值操作符属于二元中缀操作符，这说明操作符的两侧分别有一个操作数。右操作数可以是任意的表达式，不过左操作数必须是一个合法的左值，即*lvalue*（按自然语言来理解，这表示一个合法的存储位置，如一个变量或数组中的一个位



置)。最常用的赋值操作符是简单赋值。它首先确定右边表达式的值，然后将左边的变量设置为这个值。

```
$a = $b;  
$a = $b + 5;  
$a = $a * 3;
```

注意最后一个赋值中同一个变量用到了两次：一次用于计算，另一次是为了赋值。这本身没有什么问题，不过这个操作相当常用，所以有一个相应的简写形式（这是从C借鉴来的）。如果有：

```
lvalue operator = expression
```

会像这样执行：

```
lvalue = lvalue operator expression
```

只不过左值（lvalue）不会计算两次（只有当左值的计算有副作用时，二者才有区别。不过如果确实有差别，通常也能得到你想要的结果，所以不用担心）。

例如，可以把前面的例子写为：

```
$a *= 3;
```

这读作“将\$a乘以3”。在Perl中，几乎所有二元操作符都可以这么做，甚至有些在C中并不支持而在Perl中是允许的：

```
$line .= "\n";      # 为$line追加换行符  
$fill x= 80;        # 让字符串$fill重复80次  
$val ||= "2";       # 如果$val不为"true"则设置为2
```

平均分例子<sup>注20</sup>的第10行中包含两个字符串连接，其中一个是赋值操作符。另外第18行包含一个+=。

不论使用哪一种赋值操作符，左变量的最终值将作为整个赋值表达式的值返回<sup>注21</sup>。C程序员对此不会感到奇怪，他们已经知道如何使用这个技巧将变量清0：

```
$a = $b = $c = 0;
```

你还会经常看到赋值在while循环中用作条件，如平均分例子中的第8行。

让C程序员惊讶的是，Perl中的赋值会返回具体变量作为左值，所以你甚至可以在一个语句中多次修改同一个变量。例如，可以有以下赋值：

---

注20：是不是以为我们已经把它忘了？

注21：这与很多其他语言不同（比如Pascal），在那些语言中，赋值只是一个语句，不返回任何值。之前我们说过赋值类似于一个过程，不过要记住，在Perl中甚至过程也会返回值。

```
($temp - = 32) *= 5/9;
```

这会“原地”完成从华氏度到摄氏度的转换。这也解释了为什么本章前面可以使用下面这条语句：

```
chop($number = <STDIN>);
```

让它去除\$number的最后一个值。一般来说，如果想复制某个内容，同时还想对它做些其他的处理，就可以使用这个特性。

## 一元算术操作符

如果觉得\$variable += 1还不够简短，Perl甚至从C借用了一种更为简短的方式来完成变量的自增。自增（和自减）操作符就是将变量的值加（或减）1。它们可以放在变量的任意一侧，这取决于你希望何时完成计算，参见表1-3。

表1-3：自增操作符

示例	操作符名	结果
++\$a, \$a++	自增	将\$a加1
--\$a, \$a--	自减	将\$a减1

如果把这样一个“自增”或“自减”操作符放在变量的前面，则这个变量称为预增（预减）变量，它的值会在引用变量前改变。如果把它放在变量后面，则称为后增（后减）变量，它的值会在使用变量之后才改变。例如：

```
$a = 5;      # $a赋值为5
$b = ++$a;   # $b赋值为$a自增后的值，即6
$c = $a--;   # $c赋值为6，然后$a自减为5
```

平均分例子中的第15行将分数个数加1，以便知道需要对多少个分数求平均分。它使用了后增操作符（\$scores++），不过在这里使用预增还是后增操作符并没有差别，因为表达式在void上下文中，也就是说计算这个表达式只是为了得到其副作用：将变量增1。返回的值最后会丢掉不用<sup>注22</sup>。

## 逻辑操作符

逻辑操作符也称为“短路”操作符，允许程序根据多个条件做出决定，而不必使用嵌套的if语句。它们之所以被称为短路操作符，这是因为，如果认定左参数已经提供了足够的信息可以决定整个表达式的值，它们就会跳过（短路）其右参数的计算。这不只是为了提

---

注22：优化工具会注意到这一点，将后增优化为预增操作，因为这样执行会更快一些（你不需要知道这些，不过我们希望听到这个之后会开心一些）。

高效率。完全可以依赖这种短路行为来避免执行右参数中的代码，因为你知道如果没有左参数的“保护”，则这些代码可能会出问题。在Perl中你可以说“要么去加州，要么完蛋！”但不会真的完蛋（如果你果真去了加州）。

Perl实际上有两组逻辑操作符：一组是从C借用的传统操作符，另一组是更新的（不过甚至更为传统的）超低优先级操作符（借鉴了BASIC）。如果使用得当，则这两组操作符都有助于提高可读性。如果你希望逻辑操作符比逗号更优先，则可以使用C的符号操作符，如果希望逗号优先于你的逻辑操作符，则可以采用BASIC的“单词”操作符。它们的工作通常是一样的，选择使用哪一组完全取决于个人喜好（第3章“逻辑与、或、非和异或”一节中给出了一些对比这两组逻辑操作符的例子）。由于优先级不同，这两组操作符不能互换，不过，一旦通过解析，操作符本身的行为就是一样的；优先级只影响其参数的范围。表1-4 列出了这些逻辑操作符。

表1-4：逻辑操作符

示例	操作符名	结果
<code>\$a &amp;&amp; \$b</code>	And（与）	如果\$a为false则为\$a，否则为\$b
<code>\$a    \$b</code>	Or（或）	如果\$a为true则为\$a，否则为\$b
<code>! \$a</code>	Not（非）	如果\$a不为true则为true
<code>\$a and \$b</code>	And（与）	如果\$a为false则为\$a，否则为\$b
<code>\$a or \$b</code>	Or（或）	如果\$a为true则为\$a，否则为\$b
<code>not \$a</code>	Not（非）	如果\$a不为true则为true
<code>\$a xor \$b</code>	Xor（异或）	如果\$a或\$b为true，但不同时为true，则为true

由于逻辑操作符可以“短路”，在Perl中通常用来根据条件执行代码。下面这行代码（平均分例子中的第4行）试图打开文件*grades*：

```
open(GRADES, "<:utf8", "grades") || die "Can't open file grades: $!\n";
```

如果正常打开了文件，会跳至程序的下一行代码。如果没能打开文件，则会提供一个错误消息，然后停止执行。

从字面来看，这行代码表示“要么打开*grades*，要么完蛋！”短路操作符不仅类似于自然语言，还保留了我们习惯的视觉流。重要的动作列在屏幕左边，次要的动作隐藏在屏幕右边（\$!变量包含操作系统返回的错误消息，参见第25章）。当然，这些逻辑操作符还可以用在更传统的条件构造中，如if和while语句。

## 一些数值和字符串比较操作符

比较或关系操作符会指出两个标量值（数字或字符串）的相互关系。有两种这样的操作



符：一组用于完成数值比较，另一组完成字符串比较（不论哪一种情况，首先都要对参数进行“强制转换”，使它有合适的类型）。假设有左参数和右参数分别为\$a和\$b，表1-5给出了可用的一些比较操作符。

表1-5：比较操作符

比较	数值	字符串	返回值
等于	==	eq	如果\$a等于\$b，则为true
不等于	!=	ne	如果\$a不等于\$b，则为true
小于	<	lt	如果\$a小于\$b，则为true
大于	>	gt	如果\$a大于\$b，则为true
小于等于	<=	le	如果\$a不大于\$b，则为true
大于等于	>=	ge	如果\$a不小于\$b，则为true
比较	<=>	cmp	如果相等则为0；如果\$a大于\$b则为1；如果\$b大于\$a则为-1

最后一组操作符（<=>和cmp）与前面的操作符完全是冗余的。不过，它们在sort子例程中非常有用（见第27章）<sup>注23</sup>。

## 一些文件测试操作符

利用文件测试操作符，可以在盲目地处理文件之前先测试是否已经设置某些文件属性。当然，最基本的文件属性就是文件是否存在。例如，在作为一个新文件打开你的邮件别名文件之前，最好能知道它是否已经存在，因为如果这个文件原本已经存在，这会清除其中原有的全面内容。表1-6给出了一些文件测试操作符。

表1-6：文件测试操作符

示例	操作符名	结果
-e \$a	Exists（存在）	如果\$a中命名的文件已经存在，返回true
-r \$a	Readable（可读）	如果\$a中命名的文件可读，返回true
-w \$a	Writable（可写）	如果\$a中命名的文件可写，返回true
-d \$a	Directory（目录）	如果\$a中命名的文件是一个目录，返回true
-f \$a	File（文件）	如果\$a中命名的文件是一个常规文件，返回true
-T \$a	Text File（文本文件）	如果\$a中命名的文件是一个文本文件，返回true

注23：有些人认为这种冗余很糟糕，因为这与语言力求最小化（或正交）的思路相违背。不过Perl不是一个正交语言；它是一种“斜交”语言。我们的意思是，Perl不要求你总是走直角。有时你可能想试试沿斜角到达目的地。TMTOWTDI的核心是取捷径。而捷径的关键是程序员效率。

可以这样使用：

```
-e "/usr/bin/perl" or warn "Perl is improperly installed.\n";  
-f "/vmlinuz" and say "I see you are a friend of Linus.";
```

注意常规文件与文本文件并不相同。类似/vmlinuz的二进制文件是常规文件，但它们不是文本文件。文本文件与二进制文件对应，而常规文件与类似目录、设备等“非常规”文件对应。

Perl提供了大量文件测试操作符，很多都未在这里列出。大多数文件测试操作符都是一元布尔操作符，也就是说，这些操作符只取一个操作数（一个表示文件名或文件句柄的标量），会返回一个true或false值。有些文件测试操作符还会返回更有意思的结果，如文件的大小或“年龄”，如果需要，可以查阅第3章中“命名一元操作符和文件测试操作符”一节。

## 控制结构

到目前为止，除了一个比较大的例子外，所有例子都是线性的，我们只是按顺序执行各个语句。前面已经见过一些使用短路操作符的例子，可以导致某个命令执行（或不执行）。你当然可以写一些非常有用的线性程序（很多CGI脚本就属于这一类），不过，如果有条件表达式和循环机制，你还能写出功能更强大的程序。这些统称为控制结构。所以也可以把Perl认为是一种控制语言。

不过，要进行控制，首先必须能做出判断，你要知道真假的区别。

## 什么是真？

我们已经反复谈到过“真值”<sup>注24</sup>，也提到了某些操作符会返回一个true或false值。在继续深入讨论之前，确实需要好好解释一下这到底是什么意思。Perl对真值的处理与其他计算机语言稍有不同，不过使用一段时间后，就会发现这很有道理（实际上，我们希望你读完下面的内容之后就能有所体会）。

基本说来，Perl认为真值是自明的（self-evident）。这是一种灵活的说法，表示几乎任何事物的真值都可以计算。Perl采用实用的方式定义真值，即一个实体的真值取决于这个实体的类型。事实上，值为真（true）的情况要比非真多得多。

Perl中的真值都在标量上下文中计算。除此以外，不会做任何类型强制转换。所以对于标量能保存的各种类型的值，有以下规则：

1. 除了""和"0"外，所有字符串都为true。

---

注24：严格地说，这倒不是真的。



2. 除0以外，所有数字都为true。
3. 所有引用都为true。
4. 所有未定义值都为false。

实际上，后两个规则是由前两个规则推导得到的。任何引用（规则3）都会指向一个有地址的对象，因此会计算为一个包含该地址的数字或字符串，这绝对不会为0，因为它肯定已定义。另外任何未定义的值（规则4）总是计算为0或空串。

在某种程度上，如果你假装一切都是字符串，还可以从规则1推导出规则2。重申一次，要计算真值，实际上并不需要完成任何字符串强制转换，不过如果真有字符串强制转换，数值0会转换为字符串"0"，因此为false。而所有其他数字不会转换为字符串"0"，所以都将为true。下面来看一些例子，以便更好地理解：

```
0          # 将转换为字符串"0"，所以为false。
1          # 将转换为字符串"1"，所以为true。
10 - 10    # 10减去10等于0，这会转换为字符串"0"，所以为false。
0.00       # 等于0，将转换为字符串"0"，所以为false。
"0"        # 字符串"0"，所以为false。
""         # 这是一个空串，所以为false。
"0.00"     # 字符串"0.00"，既不是""也不是"0"，所以为true!
"0.00" + 0  # 将计算为数字0（由+强制转换），所以为false。
$a         # 这是$a的一个引用，所以为true，即使$a为false也是如此。
undef()    # 是一个返回未定义值的函数，所以为false。
```

我们在前面已经反复提到过，真值在标量上下文中计算，你可能想知道一个列表的真值会是什么。嗯，事实是，Perl中没有任何一个操作会在标量上下文中返回列表。它们都会注意到所在的是一个标量上下文，相应地会返回一个标量值，然后可以对这个标量应用以上真值规则。所以只要你能确定给定的操作符在标量上下文中返回的结果，就没有任何问题。事实上，如果数组或散列包含有元素，它们返回的标量值可以很方便地计算为true。后面还会介绍更多有关内容。

## if和unless语句

之前我们已经见过逻辑操作符可以用作条件。比逻辑操作符稍复杂一些的形式是if语句。if语句会计算一个真值条件（也就是一个布尔表达式），如果这个条件为true，则执行一个代码块：

```
if ($debug_level > 0) {
    # 出问题了，告诉用户
    say "Debug: Danger, Will Robinson, danger!";
    say "Debug: Answer was '54', expected '42'.";
}
```

代码块是组合在一起的一个或多个语句，由一对大括号括起。由于if语句要执行一个代码块，根据定义，必须有大括号。如果你了解诸如C之类的语言，会注意到这里与C语言有



些不同。在C中，如果代码块中只包含一个语句，大括号是可选的，而在Perl中大括号必不可少。

有时只是在条件满足时执行一个代码块还不够。可能还希望在该条件不满足时执行另一个代码块。当然也可以使用两个if语句，二者条件正好相反，不过Perl提供了一个更精巧的解决方案。在if的代码块后面，还可以再加另一个可选的条件，名为else，只有当真值条件为false时才会执行else后面的代码块（经验丰富的计算机程序员对此应该并不奇怪）。

有时甚至不只是两个选择。在这种情况下，可能还希望增加一个elsif真值条件，来提供另一个选择（有经验的计算机程序员可能会对“elsif”的拼法很惊讶，不过，没人会为此道歉）。

```
if ($city eq "New York") {  
    say "New York is northeast of Washington, D.C.";  
}  
elsif ($city eq "Chicago") {  
    say "Chicago is northwest of Washington, D.C.";  
}  
elsif ($city eq "Miami") {  
    say "Miami is south of Washington, D.C. And much warmer!";  
}  
else {  
    say "I don't know where $city is, sorry.";  
}
```

if和elsif子句会按顺序逐个计算，直到找到一个条件为true，或者到达else条件。如果找到某个条件为true，会执行这个子句的代码块，并跳过余下的所有分支。有时，你并不想在条件为true时做任何事情，只希望当条件为false时做某个工作。如果使用一个空if再加一个else看起来会很乱；如果使用否定的if条件，又不便于理解；毕竟，如果我们平常说“如果这不是真的，就做某件事情”，这听起来确实很怪异。在这些情况下，可以使用unless语句：

```
unless ($destination eq $home) {  
    say "I'm not going home.";  
}
```

不过没有elsunless。这通常解释为一个特性。

## given和when语句

要测试有多个候选值的情况，类似于其他语言中的switch和case语句，新版本的Perl也提供了相应的语句。不过，由于我们希望Perl更类似一种自然语言，所以将这两个语句取名为given和when（因为我们已经在代码前面加上了use v5.14，所以应该可以使用这两个语句，这个特性是5.10中增加的）。

```
#!/usr/bin/perl
```

```

use v5.14;

print "What is your favorite color? ";
chomp(my $answer = <STDIN>);

given ($answer) {
    when ("purple") { say "Me too." }
    when ("green") { say "Go!" }
    when ("yellow") { say "Slow!" }
    when ("red") { say "Stop!" }

    when ("blue") { say "You may proceed." }
    when (/w+, no \w+/) { die "AAAUUGHHHH!" }

    when (42) { say "Wrong answer." }

    when (['gray','orange','brown','black','white']) {
        say "I think $answer is pretty okay too.";
    }

    default {
        say "Are you sure $answer is a real color?";
    }
}

```

首先，`given`部分得到表达式的值，将它作为“谈话主题”，使`when`语句知道要测试哪个值。然后逐个计算各个分支，将各个`when`的参数与主题匹配，找出第一个与主题值匹配的`when`语句。这里会按顺序逐个匹配`when`语句，一旦找到一个匹配语句，就不再尝试后面的语句，而是直接跳出整个`given`构造。

各个`when`参数的形式（如“red”、42、`/w+, no \w+/`）确定了要完成何种类型的匹配，所以字符串会作为字符串匹配，数字作为数字匹配，模式当然要作为模式匹配。对于值列表，只要其中任何一个值匹配，这个列表就匹配。`when`语句使用了一个称为“智能匹配”的底层操作，这是为大多数可能的匹配而设计的特性（也有例外情况）。有关的更多内容请参见第3章中的“智能匹配操作符”一节。

## 循环构造

循环语句允许Perl程序反复地执行相同的代码，所以它们通常也称为迭代（iterative）构造。有多种不同类型的循环构造，它们的主要区别在于循环何时结束，从而能继续做其他处理。

### 条件循环

`while`和`until`语句首先测试一个表达式的真值，这类似于`if`和`unless`语句，不过每次只要条件满足，就会重复地执行代码块。这个条件总在每次迭代之前检查。如果条件满足（也就是说，如果`while`的条件为`true`，或`until`的条件为`false`），就会执行循环语句的代码块。

```

print "How many tickets have we sold so far? ";
my $before = <STDIN>;

my $sold = $before;
while ($sold < 10000) {
    my $available = 10000 - $sold;
    print "$available tickets are available. How many would you like: ";
    my $purchase = <STDIN>;
    if ($purchase > $available) {
        say "Too many! Try again.";
        $purchase = 0;
    }
    $sold += $purchase;
}

say "This show is sold out, please come back later.";

```

需要说明，如果初始条件未满足，就根本不会进入循环。例如，如果10 000张票已经全部售出，我们就要赶快发出通告告诉大家演出票已经告罄。

在前面的平均分例子中，第8行代码为：

```
while (my $line = <GRADES>) {
```

这里将下一行内容赋给变量\$line，前面已经解释过，这会返回\$line的值，使得while语句的条件能够计算\$line的真值。你可能想知道，如果输入为空行，则Perl是否会得到一个false值而过早地退出循环。答案是否定的。如果你想想我们之前所说的，就会找到原因。行输入操作符会在字符串末尾留一个换行符，所以即使是空行也会有一个值"\n"。而且你知道的，"\n"不属于前面规定的false值。所以这个条件为true，即使遇到空行这个循环也会继续。

另一方面，最后到达文件末尾时，行输入操作符会返回未定义值，将会计算为false。正如我们希望的，循环将终止。在Perl中没有必要显式地测试eof函数，因为输入操作符的设计很合理，可以在条件上下文中平滑地工作。

实际上，几乎所有特性都设计为可以在条件（布尔）上下文中平滑地工作。如果你在标量上下文中提到一个数组，则会返回这个数组的长度。所以你会经常看到像下面这样处理命令行参数：

```

while (@ARGV) {
    process(shift @ARGV);
}

```

shift操作符通过循环每次从参数表中删除一个元素（并返回这个元素）。数组@ARGV处



理完后，循环会自动退出；也就是说，当数组长度为0时，循环将终止。0在Perl中就是false。从某种意义上讲，这表示数组本身变为“false”<sup>注25</sup>。

## 三部分循环

另一个迭代语句是“三部分”循环，也称为C风格的for循环。三部分循环运行时与前面的while循环很类似，不过看起来有点不同，因为有两个语句被移到循环的定义中（不过C程序员会发现这很熟悉）。

```
print "How many tickets have we sold so far? ";
my $before = <STDIN>;

for (my $sold = $before; $sold < 10000; $sold += my $purchase) {
    my $available = 10000 - $sold;
    print "$available tickets are available. How many would you like: ";
    $purchase = <STDIN>;
    if ($purchase > $available) {
        say "Too many! Try again.";
        $purchase = 0;
    }
}

say "This show is sold out, please come back later.";
```

在这个三部分循环的括号里，共有3个表达式（它也因此得名），由两个分号分隔。第一个表达式设置循环变量的初始状态。第二个表达式是测试循环变量的条件；这就类似于while语句的条件。第三个表达式将修改循环变量的状态；这个表达式会在每次迭代的最后执行，相当于前面while循环中显式地修改变量状态。

这个三部分循环开始时，首先设置初始状态，然后检查真值条件。如果条件为true，则执行代码块。代码块执行结束时，将执行修改表达式（即第三个表达式），然后再次检查真值条件，如果为true，则基于这个新变量值再次运行代码块。只要真值条件一直为true，就会继续执行代码块和修改表达式（注意，只有中间的表达式需要计算求值。计算第一个和第三个表达式的目的只是为了得到其副作用，这两个表达式得到的结果值会被丢掉）。

这3个表达式都可以忽略，不过两个分号是必不可少的。如果省略了中间表达式，会认为你希望一直循环下去，所以可以写一个无限循环，如下所示：

```
for (;;) {
    say "Take out the trash!";
}
```

---

注25：这是Perl程序员的看法。所以没有必要将0与0比较，看它是否为false。尽管其他语言可能强制要求你这么，但在Perl中大可不必，不用花功夫去写这种显式比较（如while(@ARGV != 0）。不论是对你还是对计算机，这都是低效的做法。对于将来要维护代码的人来说也是如此。

```
    sleep(5);  
}
```

## foreach循环

Perl中最后一种迭代语句名为`foreach`循环<sup>注26</sup>。这个循环会针对一个已知的标量列表反复执行相同的代码，例如可以从一个数组得到这样一个列表：

```
for my $user (@users) {  
    if (-f "$home{$user}/.nexrc") {  
        say "$user is cool... they use a perl-aware vi!";  
    }  
}
```

`if`和`while`语句会为条件表达式提供标量上下文，与之不同，`foreach`语句会为括号中的表达式提供列表上下文。所以会计算这个表达式来生成一个列表（如果不能生成列表，而只得到一个标量值，可以认为生成了一个只包含单个元素的列表）。然后依次将列表的各个元素赋至循环变量，针对列表的每一个元素都将执行一次代码块。需要说明，循环变量指示的是元素本身，而不是元素的一个副本。所以，修改这个循环变量同时也会修改原始数组。

你会发现，在典型的Perl程序中，这种循环远远多于三部分的`for`循环，因为在Perl中很容易生成`foreach`循环迭代处理的那些列表（部分出于这个原因，我们窃取了`for`的关键字，因为我们很懒，认为常用的词应该更简短）。你会经常看到以下做法，由一个循环迭代处理已排序的散列键：

```
for my $key (sort keys %hash) {
```

实际上，我们的平均分例子中第13行就是这样做的，以便按字母顺序打印出学生的平均分。

## 跳出循环：next和last

`next`和`last`操作符允许你改变循环控制流。经常会有特殊情况出现；你可能想跳过这种特殊情况，或者也可能希望在遇到这种特殊情况时退出循环。例如，如果你在处理UNIX帐户，可能想跳过系统帐户（如`root`或`lp`）。`next`操作符允许你直接跳到本次循环迭代的末尾，并开始下一次迭代。`last`操作符则允许你跳到代码块的末尾，就好像循环的测试条件返回`false`一样。有些情况下这可能很有用，例如，你在寻找一个特定的帐户，希望一旦找到就退出。

---

注26：追溯历史，写这个语句时有一个`foreach`关键字，所以由此得名。如今我们更倾向于使用`for`关键字，因为加入一个`my`声明时它读起来更像英语（另一个原因是，其语法不会与三部分循环混淆）。所以我们很多人不再写`foreach`，当然如果你愿意，继续写`foreach`也是可以的。

```

for my $user (@users) {
    if ($user eq "root" || $user eq "lp") {
        next;
    }
    if ($user eq "special") {
        print "Found the special account.\n";
        # 完成一些处理
        last;
    }
}

```

通过为循环设置标签，并指定希望跳出哪个循环，还可以跳出多层循环。结合语句修饰符（这是另一种形式的条件，后面还会详细介绍），可以很清楚地写出退出循环的代码，这有很好的可读性（假设你认为英语很好懂）：

```

LINE: while (my $line = <EMAIL>) {
    next LINE if $line eq "\n";           # 跳过空行
    last LINE if $line =~ /^>/;          # 在第一个引号行停止
    # 这里是你的内容
}

```

你可能会说，“等一下，那两个斜线中间有一个奇怪的`>`，那是什么东西？看起来可不像英语。”你说的没错。这是一个包含正则表达式的模式匹配（不过相当简单）。这正是下一节要讨论的内容。Perl可能是世界上最棒的文本处理语言，而正则表达式就是Perl文本处理的核心。

## 正则表达式

正则表达式（Regular expressions）也称为`regexe`、`regexp`或`RE`，广泛应用于很多搜索程序（如`grep`和`findstr`）、文本处理程序（如`sed`和`awk`）和编辑器（如`vi`和`emacs`）。正则表达式是一种描述一组字符串的方法，而无需逐个列出所有这些字符串<sup>注27</sup>。很多其他计算机语言也采纳了正则表达式（其中一些甚至标榜为“Perl5正则表达式”），不过所有这些语言都没有像Perl那样将正则表达式真正集成到语言中。在Perl中，正则表达式有很多用法。首先，正则表达式可以在条件语句中使用，以确定一个字符串是否与一个特定模式匹配，因为在一个布尔上下文中正则表达式会返回`true`和`false`。所以如果你在一个条件语句中看到类似`/foo/`的代码，应该知道你看到的是一个标准的模式匹配操作符：

```

if (/Windows 7/) { print "Time to upgrade?\n" }

```

其次，如果能够在字符串中找到模式，还可以把它们替换为其他内容。所以，如果看到类似`s/foo/bar/`的代码，应该知道，这是在要求Perl用“bar”来替换“foo”（如果

---

注27：关于正则表达式的概念有一个很好的资源，可以参考Jeffrey Friedl编写的《Mastering Regular Expressions》。



可以)。我们将它称为替换 (substitution) 操作符。它还会根据替换是否成功返回true或false, 不过执行替换操作符通常只是为了得到其副作用:

```
s/IBM/lenovo/;
```

最后, 模式不仅能指定哪里有数据, 还能指定哪里不是数据。所以split操作符使用一个正则表达式来指定哪里不是数据。也就是说, 正则表达式定义了界定数据字段的分隔符 (separator)。我们的平均分例子中就有这样的两个小例子。第9行和第16行分别利用空白符分隔字符串, 从而返回一个单词列表。不过还可以根据用正则表达式指定的任何操作符划分字符串:

```
my ($good, $bad, $ugly) = split(/,/ , "vi,emacs,teco");
```

这些情况下还可以使用一些修饰符来完成一些特殊的处理, 如匹配字母字符时可以忽略大小写, 不过这些细节将在第2部分中再做介绍。

正则表达式最简单的用法是匹配一个直接量表达式。在前面的split中, 我们匹配的是单个逗号字符。不过如果要连续匹配多个字符, 必须按顺序匹配。也就是说, 模式要查找一个子串, 这正是你想要的。假设我们希望显示一个HTML文件 (其中包含HTTP链接, 而不是FTP链接) 中的所有行。假设我们第一次处理HTML, 没有什么经验。我们知道所有链接肯定在某个位置上包含有“http:”。可以用以下代码循环处理文件:

```
while (my $line = <FILE>) {  
    if ($line =~ /http:/) {  
        print $line;  
    }  
}
```

在这里, =~ (模式绑定)告诉Perl在变量\$line中查找正则表达式“http:”的一个匹配。如果找到这个表达式, 这个操作符会返回一个true值, 并执行代码块 (一个print语句)<sup>注28</sup>。

顺便说一句, 如果没有使用=~绑定操作符, Perl会搜索一个默认字符串而不是\$line。就好像当你说, “哎呀! 帮我找找我的镜头!” 人们自然知道要在你的周围寻找, 而不需要你明白地告诉他们。类似地, Perl知道, 如果你没有指出要在哪里搜索, 则要在一个默认字符串中搜索你要找的内容。这个默认字符串实际上是一个特殊的标量变量, 有一个奇怪的名字\$\_.事实上, 这并不只是模式匹配的默认字符串; Perl中的很多操作符都默认使用\$\_变量, 所以经验丰富的Perl程序员可能会把上面的例子写为:

```
while (<FILE>) {  
    print if /http:/;  
}
```

嗯, 这里又出现一个语句修饰符。奇怪的小东西。

---

注28: 这与UNIX命令grep 'http:' file 的做法非常相似。

这很方便，不过如果我们想找出所有链接类型，而不只是HTTP链接，该怎么做呢？可以提供链接类型列表，如“http:”，“ftp:”，“mailto:”等。不过这个列表可能很长，另外，增加一个新的链接类型时又该怎么办？

```
while (<FILE>) {
    print if /http:/;
    print if /ftp:/;
    print if /mailto:/;
    # 下一个是什么?
}
```

由于正则表达式是对一组字符串的描述，我们可以直接描述想查找的东西：一些字母字符，后面有一个冒号。按正则表达式的说法，这就是/[a-zA-Z]+:/，中括号定义了一个字符类（class）。a~z和A~Z表示所有ASCII字母字符（短横线表示起始字符和结束字符之间的所有字符，包括起始和结束字符）。+ 是一个特殊字符，表示“我前面的内容出现一次或多次”。我们把它称为一个量词（quantifier），表示允许重复多少次（斜线实际上并不是正则表达式的一部分，而是属于模式匹配操作符。斜线就相当于引号，只不过其中包含的是一个正则表达式）。

由于有些字符类（如字母）相当常用，Perl为它们定义了简写形式，如表1-7所列。

表1-7：字母字符的简写

名字	ASCII定义	Unicode定义	简写
空白符	[ \t\n\r\f]	\p{Whitespace}	\s
单词字符	[a-zA-Z_0-9]	[\p{Alphabetic}\p{Digit}\p{Mark}\p{Pc}]	\w
数字	[0-9]	\p{Digit}	\d

需要指出，这些只匹配单个字符。 \w会匹配单个单词字符，而不是整个单词（还记得+量词吗？可以用 \w+匹配一个单词）。通过使用大写字符，Perl还提供了这些字符类的反集，如 \D表示非数字字符。

需要说明， \w并不完全等价于 [a-zA-Z\_0-9]（ \d也不总是[0-9]）。有些本地化环境（locale）在ASCII序列之外还定义其他字母字符， \w会涵盖这些字符。5.8.1以上的Perl版本还支持Unicode字母和数字属性，并利用这些属性相应地处理Unicode字符（Perl还认为象形文字和组合标记都可以归入 \w字符）。

还有另外一种非常特殊的字符类，写作“.”，它能与任何字符匹配<sup>注29</sup>。例如， /a./将匹

注29： 只有一个例外，正常情况下它不能匹配换行符。可以想想看，在grep(1)中“.”也不能匹配换行符。



配任何包含“a”且满足以下条件的字符串，即“a”不能是字符串中的最后一个字符。因此，它能匹配“at”或“am”，甚至可以匹配“a!”，但不能匹配“a”，因为“a”后面没有字符来与点号匹配。由于它会在字符串中的任何位置搜索模式，所以能匹配“oasis”和“camel”，但不能匹配“sheba”。它会在“caravan”的第一个“a”处匹配。本来在第二个“a”处也可以匹配，不过在从左到右搜索并找到第一个合适的匹配后，它会停止搜索。

## 量词

前面讨论的字符和字符类都与单个字符匹配。我们提到过，还可以用`\w+`匹配多个“单词”字符。`+`是一种量词，除此之外还有其他一些量词。这些量词都放在需要“定量”的内容后面。

量词的最一般的形式是同时指定某一项可以匹配的最小和最大次数。这两个数要放在大括号里，用一个逗号分隔。例如，如果想匹配北美的电话号码，序列`\d{7,11}`将至少匹配7位数字，但不能超过11位数字。如果在大括号中只放了一个数字，这个数会同时指定最小和最大次数；也就是说，这个数明确指定了某一项可以匹配的次数（未加量词的项都有一个隐式的`{1}`量词）。

如果指定了最小次数和逗号，不过没有指定最大次数，最大次数则为无限大。换句话说，它要至少匹配指定的最小次数，然后可以重复无限次。例如，`\d{7}`只匹配前7位数字（如，本地的北美电话号码，或者一个更长号码的前7位），而`\d{7,}`将匹配任何电话号码，甚至国际电话号码（除非它少于7位数字）。要指出“最多”多少次，并没有特殊的方式。例如，`{0,5}`就可以用来查找最多5个任意字符。

有些最小和最大次数的组合频繁出现，所以Perl还为此定义了特殊的量词。我们已经见过`+`，这相当于`{1,}`，或“前面的项至少出现1次”。还有一个`*`，这与`{0,}`相同，或“前面的项出现0次或多次”。另外量词`?`等价于`{0,1}`，表示“前面的项出现0次或1次”（也就是说，前面的项是可选的）。

关于定量，有两点要特别注意。首先，Perl量词默认是“贪婪的”。这意味着，只要整个模式仍匹配，Perl量词总是试图与尽可能多的字符匹配。例如，如果将`/\d+/`与“1234567890”匹配，它会匹配整个字符串。使用“.”（任意字符）时尤其要注意这一点。人们经常会写类似这样的字符串：

```
larry:JYHtPh0./NJTU:100:10:Larry Wall:/home/larry:/bin/bash
```

试图将“larry:”与`/.+:/`匹配。不过，由于`+`量词是贪婪的，这个模式会一直匹配到“/home/larry:”，因为它会匹配尽可能多的字符，直到最后一个冒号，这包括所有其他冒号。有时通过使用一个非字符类可以避免这种情况；也就是说，通过指定`/[^:]+:/`（这



表示要尽可能多地匹配1个或多个非冒号字符），则只会匹配到第1个冒号。这里的小^符号“反置”了字符类的布尔含义<sup>注30</sup>。另外要注意的一点是，正则表达式总是力图尽早匹配。这一点甚至比贪婪性更优先。由于是从左向右扫描，模式会尽可能从左边最远处匹配，即使在其他位置上可能可以匹配更长的字符串（正则表达式是贪婪的，不过它们绝不拖延）。例如，假设你在默认字符串（也就是变量\$\_）上使用替换命令(s///)，希望从字符串中间删除一个x串。如果有：

```
$_ = "fred xxxxxx barney";  
s/x*//;
```

这不会有任何效果！这是因为，x\*（表示“x”字符出现0次或多次）将与字符串最前面的“空”匹配，因为空串宽度就是0个字符，而且在“fred”的“f”前面显然有一个空串<sup>注31</sup>。还有一点你要知道。默认的，量词会应用到它前面的单个字符，所以/bam{2}/将匹配“bamm”，但不匹配“bambam”。要想为多个字符应用量词，这需要使用小括号。所以如果要匹配“bambam”，就要使用模式bam){2}/。

## 最小匹配

如果你使用的是一个远古级版本的Perl，而且不想使用贪婪匹配，就必须使用非字符类（实际上，这也是一种受限的贪婪匹配）。

在现代版本的Perl中，可以强制非贪婪匹配，即最小匹配，只需要在量词后面加一个问号。现在我们的用户名匹配将是/.\*?:/。这里的.\*? 将尝试匹配尽可能少的字符，而不是尽可能多，所以它会在第1个冒号处停下来，而不是一直匹配到最后一个冒号。

## 锁定位置

想要匹配一个模式时，会在每一个位置进行匹配，直到找到一个匹配为止。锚（anchor）允许你限制模式匹配的位置。基本说来，锚匹配的不是具体事物，这是一种特殊的无形的东西，取决于它的周围环境。也可以把它叫做规则、约束或断言。不论你怎么称呼它，锚会尝试匹配一个宽度为0的东西，可能成功，也可能失败（失败只是意味着模式不能匹配这种特殊方式。但是如果还有其他方式可以尝试，模式还会继续尝试匹配其他方式）。

特殊符号\b匹配一个单词边界，这定义为一个单词字符(\w)和一个非单词字符(\W)之间的“无形的东西”（单词字符和非单词字符的前后顺序是任意的）。字符串开头和结尾不存在的字符可以认为是非单词字符。例如：

```
/\bFred\b/
```

---

注30：抱歉，我们没有选用这种记法，不要责怪我们。这只是UNIX文化中非字符类的通常写法。

注31：别难过。甚至本书作者有时也会犯这样的错误。

可以匹配“The Great Fred”和“Fred the Great”中的“Fred”，但不能匹配“Frederick the Great”中的“Fred”，因为“Frederick”中的“d”后面不是一个非单词字符。

类似的，字符串的开头和结尾也有锚。如果^符号是模式的第一个字符，则它会匹配字符串开头的“无形字符”。因此，模式/^Fred/能匹配“Frederick the Great”中的“Fred”，但不能匹配“The Great Fred”中的“Fred”，而/Fred^/与二者都不能匹配（实际上，这个模式根本没有意义）。美元符（\$）与^符号类似，只是它与字符串末尾（而不是开头）的“无形字符”匹配<sup>注32</sup>。所以现在你可能已经明白，如果有以下代码：

```
next LINE if $line =~ /^#/;
```

这是指“如果这一行以一个#字符开头，就进入LINE循环的下一迭代。”

之前我们说过，序列\d{7,11}将匹配一个7到11位的数字。严格地讲，这句话有些误导：在实际的模式匹配操作符中使用这个序列时，如/\d{7,11}/，除了11位匹配的数字外，这个模式不会排除后面额外的不匹配数字！通常需要在定量模式的一端或两端设置锚才能得到你真正想要的结果。

## 反向引用

前面提到过，可以使用小括号为字符分组，从而为整组字符指定量词，另外还可以使用小括号记住所匹配的具体内容。用一对小括号括起正则表达式的一部分，这会记住这一部分所匹配的内容，以备以后使用。它不会改变这部分匹配的内容，所以/\d+/和/(\d+)/仍会匹配尽可能多的数字，不过，对于后者，所匹配的结果将记在一个特殊变量中，以便以后反向引用。

如何反向引用所记住的部分字符串，这取决于你希望从哪里开始。在同一个正则表达式中，可以使用一个反斜线，后面是一个整数。这个整数对应一对给定的小括号，由相应的左括号数确定（从模式开头数起，起始为1）。所以，如果要匹配类似HTML标记的字符串（如“<B>Bold</B>”），可以使用/<(.\*?)>.\*?<\1>/。这要求模式的两部分匹配同一个字符串，如此例中的“B”。

在正则表达式之外，如替换操作符的替换部分，可以使用一个\$，后面跟一个整数；这是由这个整数命名的一个正常的标量变量。例如，如果想交换一个字符串的前两个单词，则可以使用以下模式：

```
s/(\S+)\s+(\S+)/$2 $1/
```

---

注32：这有点过于简化了，因为这里假设你的字符串中不包含换行符；^和\$实际上是行（而不是字符串）开头和末尾的锚。我们将在第5章（尽可能）澄清这个问题。



替换操作符的右边（第二个和第三个斜线之间）基本上是一个双引号字符串，所以可以在这里内插变量，包括反向引用变量。这是一个很强大的概念：内插（在受控情况下）是Perl成为优秀的文本处理语言的原因之一。当然，另一个重要原因就是模式匹配。正则表达式擅长分解，而内插擅长把它们再合并起来（毕竟，鸡蛋碎了可能也有希望复原）。

如果厌倦了编号的反向引用，v5.10和更高版本还支持命名反向引用。下面的替换与前面一样，不过这里使用了命名组（见表1-8）：

```
s/(?<alpha>\S+)\s+(?<beta>\S+)/${beta} ${alpha}/
```

表1-8：正则表达式反向引用

位置	编号组	命名组
声明	( ... )	(?<NAME> ... )
同一个正则表达式中	\1	\k<NAME>
正常Perl代码中	\$1	\${NAME}

使用命名反向引用时，输入代码可能更长一些，不过随着模式的规模和复杂性不断增加，你会很庆幸可以用有意义的单词来命名组，而不只是编号。

## 列表处理

本章较早前提到过，Perl有两个主要的上下文：标量上下文（用于处理单数事物）和列表上下文（用于处理复数事物）。目前为止介绍过的很多传统操作符都严格执行标量操作。它们取单数参数（对于二元操作符，则取一对单数参数），而且总会生成一个单数结果（甚至在列表上下文中也不例外）。如果有以下代码：

```
@array = (1 + 2, 3 - 4, 5 * 6, 7 / 8);
```

可以知道，右边的列表包含4个值，因为算术操作符总是生成标量值，即使是在列表上下文（由数组赋值提供）中也是如此。

不过，其他Perl操作符可能生成标量值，也可能生成列表值，这取决于其上下文。它们当然“知道”将会得到一个标量还是一个列表。不过，你怎么知道呢？实际上，知道这一点很容易，只需要仔细考虑几个关键的概念。

首先，列表上下文肯定是由“周围环境”中的某个东西提供的。在前面的例子中，列表赋值会提供列表上下文。之前我们还看到过foreach循环的列表提供了列表上下文。print操作符也能提供列表上下文。不过你不用逐个地去学习。

在本书后面，我们将经常给出一些语法概要，你会看到一些操作符定义为取一个LIST为参数。这些就是提供列表上下文的操作符。在本书中，LIST用作一个特定的技术术语，表示



“一个提供列表上下文的语法构造”。例如，如果查找`sort`，则可以找到它的语法概要：

```
sort LIST
```

这说明`sort`会为其参数提供列表上下文。

其次，在编译时（也就是说，Perl解析你的程序，并翻译为内部操作码），任何取`LIST`参数的操作符都会向该`LIST`中的各个语法元素提供列表上下文。所以在编译时，`LIST`中的每一个顶级操作符或实体都知道它要生成一个列表。这说明，如果有以下代码：

```
sort @dudes, @chicks, other();
```

`@dudes`、`@chicks`和`other()`在编译时都知道它要生成一个列表值而不是一个标量值。所以编译器会生成相应的内部操作码来反映这一点。

接下来，在运行时（具体解释内部操作码时），这些`LIST`元素依次生成列表，然后（这一点很重要）所有单个列表会首尾相接，合并为一个列表。这个“压扁”的一维列表最后会交给原先希望得到`LIST`的函数。所以，如果`@dudes`包含(Fred,Barney)，`@chicks`包含(Wilma,Betty)，另外`other`函数返回单元素列表 (Dino)，那么`sort`看到的`LIST`就是：

```
(Fred,Barney,Wilma,Betty,Dino)
```

而`sort`返回的`LIST`为：

```
(Barney,Betty,Dino,Fred,Wilma)
```

有些操作符会“生产”即生成列表（如`keys`），而有些操作符会“消费”即处理列表（如`print`），还有一些可以把列表转换为其他列表（如`sort`）。最后这一类操作符可以认为是过滤器，只不过与shell中不同，数据流是从右向左，因为列表操作符会从右处理传入的参数。可以连续“叠加”多个列表操作符：

```
print reverse sort map {lc} keys %hash;
```

这会得到`%hash`的键，然后返回给`map`函数，这个函数对各个键应用`lc`操作符，将所有键改为小写，再把它们传递给`sort`函数，`sort`函数对这些键进行排序，把它们传递给`reverse`函数，由它逆置列表元素的顺序，最后传递到`print`函数，打印所有元素。

可以看到，用Perl描述比用自然语言描述容易多了。

列表处理还可以采用很多其他方式生成更自然的代码。这里无法一一罗列，不过作为一个例子，下面再来看看正则表达式。我们说过，可以在标量上下文使用模式查看是否匹配，不过如果在列表上下文使用模式，则会做另外一件事：它会取出所有反向引用，作为一个列表。假设你在搜索一个日志文件或一个邮箱，希望解析一个包含某种时间格式（“12:59:59 am”）的字符串。可以写作：

```
my ($hour, $min, $sec, $ampm) = /(\d+):(\d+):(\d+) *(\w+)/;
```

这种方法很方便，可以同时设置多个变量。不过也可以简单地写作：

```
my @hmsa = /(\d+):(\d+):(\d+) *(\w+)/;
```

这里会把所有4个值放在一个数组中。让人奇怪的是，从Perl表达式分离正则表达式的能力后，列表上下文更能增强Perl语言的能力。我们不常承认这一点，不过Perl除了作为一种斜交语言外，实际上也是一种正交语言。尽情享受吧。

## 有些内容你不知道也没有（太大）危害

最后，请允许我们再来强调Perl作为一个自然语言是什么概念。讲自然语言的人可以有不同的水平，可以讲语言的不同子集，可以边学边用，通常在了解语言的全部内容之前就可以很好地使用这个语言。你还没有完全了解Perl，就像你还没有参透英语一样。不过，在Perl文化中，这是完全可以的。你可以很好地使用Perl，尽管我们甚至还没有告诉你如何编写你自己的子例程。我们几乎没有解释如何把Perl看作是一个系统管理语言，或一个快速原型语言，或者一个网络语言，或是一个面向对象语言。如果写这些内容，可能需要写好几章（想想以前的版本，原先我们确实是这样做的）。

不过，最终你还是要对Perl建立自己的看法。你会痛并创作着，这是你作为艺术家的特权。我们可以告诉你我们怎么画画，但是你该怎么画我们可教不了。要知道，条条大路通罗马（TMTOWTDI）。

尽享其中的乐趣吧。





# 细节详述



# 集腋成裘

我们先从小处着手，所以这一章将讨论Perl的元素。

既然是从小处开始，可以想见，接下来几章肯定是从小到大逐步展开。也就是说，我们将采用一种自底向上的方法，先介绍Perl程序中最小的组件，再用它们构建更复杂的结构，就像分子由原子构成一样。这种做法有一个缺点，你会直接陷入到细节当中，而无法对全局有个整体的认识。不过，这样做也有一个好处，你会在学习过程中逐步理解我们给出的例子（如果你更喜欢自顶向下的方式，可以把这本书倒过来，从最后一章开始往前读）。

每一章都建立在前一章基础之上（如果你是反着读的，那么每一章都以下一章为基础），所以如果你喜欢跳着读，一定要特别当心。

在学习过程中，欢迎你随时翻阅本书最后给出的参考资料（这可不算是我们厌恶的那种“四处打听”）。特别需要指出，用等宽字体显示的所有单词基本上都会在第27章列出。尽管我们想努力做到操作系统中立，但是如果你不熟悉UNIX术语，假设遇到一个词与你设想的含义不一致，可以看看术语表中有没有这个词。如果在术语表中找不到，很可能在最后的索引中可以找到。如果还是找不到，那就用你喜欢的搜索引擎在网上搜搜看。

## 原子

尽管在底层有很多不可见的东西（稍后就会解释），不过Perl中处理的最小元素通常是单个字符。没错，就是字符；历史上，Perl总是把字节和字符混为一谈，将字节当成字符，而把字符当成字节，不过在这个全球网络化的新时代，一定要注意区别这两个概念。

当然，Perl也可以完全用7位ASCII字符集编写。出于历史的原因，128~255的字节在Perl看来都来自ISO-8859-1 (Latin1)字符集，其码点与Unicode对应。要想告诉Perl当前源文件中的字节要处理为UTF-8编码的Unicode，需要在文件最前面增加以下声明：



```
use utf8;
```

第6章还会介绍，Perl从新千年以来就提供了Unicode支持。这个支持贯穿整个语言：可以在标识符（变量名等）中使用Unicode字符，也可以在直接量字符串中使用。使用Unicode时，不用担心表示一个字符需要多少位或多少字节。Perl会假装所有字符大小都相同（也就是大小为1），尽管给定的字符在内部可能要用多字节表示。正常情况下，Perl在内部将字符表示为UTF-8，这是一个变长编码（例如，笑脸字符☺的Unicode为U+263A，在内部会表示为一个3字节的序列，不过你不用考虑这些具体细节）。

如果进一步与物理元素相对照，可以认为字符具有原子性，就像是不同元素中的各个原子。没错，字符由位和字节这些更小的“粒子”组成，不过如果分解一个字符（当然是在字符加速器中），单个的位和字节不具有整个字符的特定化学属性。就像中子是U-238原子的实现细节一样，字节相当于U+263A字符的实现细节。

所以不要担心这些小东西。下面来看更大、更好的东西。

## 分子

Perl是一种形式自由（free-form）的语言，不过这并不表示Perl完全没有形式。正如计算机工作者们经常所说，形式自由的语言是指，你可以在你喜欢的任何地方放置空格、制表符和换行符（除了不能放的地方以外）。

显然，不能把空白符放在token中间。我们用token表示一个有单位含义的字符序列，就像自然语言中的一个单词。不过与一般的单词不同，token中除了字母外，还可以包含其他字符，只要它们共同构成一个单位含义（从这个意义上讲，token更像是分子，分子不一定仅由某一种原子组成）。例如，数字和数学操作符就可以认为是token。标识符（identifier）是一个以字母字符（通常是一个字母）或连接符号（如下划线）开头的token，而且只包含字母、组合标记、数字和下划线。token不能包含空白符，因为这会将token分为两个token，就像如果英语单词中有一个空格，这会把它变成两个单词。<sup>注1</sup>

尽管两个token之间允许有空白符，但是只有当两个token放在一起会被误以为是一个token时才有必要在它们之间加空白符，以避免误解。如果作为这个目的，所有空白符都是等价的。只有在引号字符串、格式和某些面向行的引用（quote）中，换行符与空格和制表符才有区别。具体地，换行符不能像在某些其他语言中那样（如Fortran或Python）终止语句。Perl中的语句要用分号终止，就像C以及它的各种派生语言（如C++和Java）中一样。

Unicode空白符可以出现在Unicode Perl程序中，不过要特别小心。如果使用了特殊的

---

注1： 聪明的读者会指出，直接量字符串就可以包含空白符。不过字符串中可以有空白符是因为它的两端有引号，可以把空格包住（不会漏出去）。

Unicode段落和行分隔符，要当心Perl统计的行号可能与文本编辑器统计的行号不同，所以错误消息可能更难解释。最好一直使用老式的换行符。

识别token时采用的是一种贪婪方式；如果特定情况下Perl解析器要在识别一个短token还是一个长token之间做出选择，它肯定会选择长的那一个。如果你想表示两个token，只需要在它们之间插入一些空白符（我们喜欢在大多数中缀操作符两边加一些额外的空格，这只是为了便于阅读）。

注释由#字符指示，从这个#字符一直到行尾都将是注释。注释依靠空白符来分隔token。对于你在注释里放置的内容，Perl语言不会附加任何特殊含义：<sup>注2</sup>

```
my $comet = 'Haley'; # 这是一个注释
```

还有一点比较特别，如果某一行以=开头，这个语句是合法的，从这一行向下，直到下一个以=cut开头的行，这之间的所有内容都会被Perl忽略。被忽略的这些文本称为pod或“无格式旧式文档”（“plain old documentation”）。Perl的发布版本提供了一些程序，可以从Perl模块抽取pod注释，并把它转换为纯文本、手册页、L<sup>A</sup>T<sub>E</sub>X，甚至可以转换为HTML或XML文档。与之对应，Perl解释器可以从Perl模块抽取Perl代码，而忽略pod。所以可以把它看作是另外一种多行注释。甚至pod中的代码不会进行编译：

```
=pod

my $dog = 'Spot';
my $cat = 'Buster';

=cut
```

你可能认为这没有任何意义，不过以这种方式建立文档的Perl模块能保持其文档永远是最新的。关于pod的更多细节，请参见第23章，其中还介绍了如何在Perl中建立多行注释。

不过不要小看平常的注释字符。看到多行注释左边一列#字符整齐排开，会有一种很舒服的视觉效果。它会直观地告诉你：“这不是代码。”要知道，甚至在提供多行注释机制的语言中（如C），人们仍然经常在注释左边加上一列\*字符。尽管看似不重要，但实际上外观确实非常重要：

```
# 多行注释的开头
# my $dog = 'Spot';
# my $cat = 'Buster';
```

在Perl中，与在化学和语言中一样，可以利用小的结构构建越来越大的结构。我们已经提

---

注2：实际上，这里撒了一个小谎。Perl解析器会在第一个#!行中寻找命令行开关（参见第17章）。它还会解释很多预处理器生成的行号指令（参见第21章“其他语言中生成Perl”一节）。有些分子（如Perl::Critic和Smart::Comments）还可以使用特殊的注释来确定要做什么。



到了语句。语句（statement）就是一系列token，它们组成了一个命令；也就是说，这是一个祈使句。可以把一系列语句组合到用大括号（braces）包围的一个代码块（block）中[有些人分不清“braces”和“suspenders”（都有弯钩的意思），所以有意把大括号称为“花括号”]。代码块可以进一步结合形成更大的代码块。有些代码块将作为子例程（subroutines），它们可以结合成模块（modules），而模块可以进一步组合，最后形成程序（programs）。不过我们有点扯远了，这些是后面章节要讨论的主题。下面先基于字符构建一些token。

## 内置数据类型

在讨论由字符构建的各种token之前，还需要几个更抽象的概念。具体来讲，我们需要3个数据类型。

不同的计算机语言提供的数据类型数量和种类都不相同。一些常用语言为类似的值提供了很多让人混淆的数据类型，与这些语言不同，Perl只提供了几种内置数据类型。可以看看C，在C中你会遇到char、short、int、long、long long、bool、wchar\_t、size\_t、off\_t、regex\_t、uid\_t、u\_longlong\_t、pthread\_key\_t、fp\_exception\_field\_type5等类型。这还只是它的一些整数类型！此外还有浮点数、指针和字符串（想想就让人头疼）。

所有这些复杂的类型都对应到Perl中的一种类型：标量（一般来讲，Perl的简单数据类型就足以满足你的全部需要了，不过如果还不够，完全可以使用Perl的面向对象特性来定义更多有趣的动态类型，参见第12章）。Perl的三种基本数据类型为：标量、标量数组和标量散列（也称为关联数组）。有些人可能喜欢把它们称为数据结构（data structure），而不是数据类型。这也是可以的。

标量是最基本的类型，可以由标量构建更复杂的结构。标量存储单个简单值，通常是一个字符串或一个数字。可以结合这种简单类型的元素来形成另外两种聚合类型。数组（array）是一个有序的标量列表，可以用一个整数下标（或索引）来访问。Perl中的所有索引都从0开始。不过，与很多编程语言不同，Perl认为负下标也是合法的：如果下标为负值，这表示不是从头开始计数，而是从要索引的数组末尾反向计数（除了常规的下标，这也适用于各种求子串和求子列表的操作）。另一方面，散列（hash）则是包含键/值（key/value）对的一个无序的集合，访问散列时，要用字符串（键）作为下标来查找与给定键对应的标量（值）。变量的类型无非这3种。除了变量，Perl还提供了另外一些抽象（你可能认为它们也是数据类型），如文件句柄、目录句柄、格式、子例程、符号表和符号表记录。

抽象是很好的概念，我们会在后面的学习过程中认识更多的抽象，不过从某个角度讲，抽象也是没有用的东西。直接用抽象什么也做不了。正因如此，计算机语言提供了语法。我们要向你介绍各个语法项，你可以用它们把你的抽象数据组合为表达式。谈到这些语法单



元时，我们喜欢用“项”（term）这个专业术语（嗯，这里的措辞看起来让人有点摸不着头脑。不过，只要记住原来数学老师如何解释等式中的“项”，你就不会弄错）。

就像数学等式中的项一样，Perl中大多数项的作用就是为操作符（如加法和乘法）生成将由它们处理的值。不过，与数学等式中不同，Perl必须用它计算的值具体做些什么，而不只是简单地考虑等式两边是否相等。对值最常见的一种处理就是把它存储在某个地方：

```
$x = $y;
```

这是赋值操作符的一个例子（而不是数值相等操作符，Perl中表示数值相等的操作符是`=`）。这个赋值将得到`$y`的值，并把这个值放在`$x`中。注意，这里用到项`$x`不是为了它的值，而是用来表示它的位置。（经过这个赋值，`$x`原来的值将修改为一个新值）。我们说`$x`是一个左值（lvalue），这表示它是一种存储位置，可以用在赋值的左边。我们称`$y`是一个右值（rvalue），因为它用在赋值的右边。

除此以外还有第三类值，称为临时值（temporary value），如果你想知道Perl究竟如何处理左值和右值，就需要理解临时值。如果要做数学计算，如：

```
$x = $y + 1;
```

Perl将得到右值`$y`，将它与右值`1`相加，这会生成一个临时值，最后这个临时值将赋给左值`$x`。可以这么说：Perl把这些临时值存储在一个称为堆栈（stack）<sup>注3</sup>的内部结构中，了解到这一点，你就能想象出Perl到底是怎么做的。表达式（这一章要讨论的内容）把值压入堆栈，表达式操作符（将在下一章讨论）则从堆栈中将值弹出，可能会在堆栈中留下另一个临时结果，由下一个操作符处理。压入和弹出要完全匹配，表达式处理完成时，堆栈将清空（或者与开始时的状态一样）。后面还会介绍更多有关临时值的内容。有些项只能是右值，如上面的`1`，而另外一些项既可以作为左值也可以作为右值。特别地，如前面的赋值所示，变量就是如此，既可以作为左值也可以作为右值。下一节就来讨论变量。

## 变量

不用奇怪，对应前面提到的3个抽象数据类型有3种变量类型。它们分别有一个前缀，我们称这些前缀为印记（sigil）<sup>注4</sup>。标量变量名字前面总有一个`$`，甚至指示数组或散列中的某个标量元素时，最前面也要有一个`$`。这有点像英语里的“the”，参见表2-1。

---

注3：堆栈的工作就像你在小餐馆里看到的那些有弹力的碗碟机，可以把盘子压到一摞盘子最上面（栈顶），也可以再把它们弹出来[用计算机科学的语言来讲，就是压入（push）和弹出（pop）]。

注4：称之为印记可能是因为它能把一个普通的名字变得更有魔力。

表2-1：访问标量值

构造	含义
<code>\$days</code>	简单标量值 <code>\$days</code>
<code>\$days[28]</code>	数组 <code>@days</code> 中的第29个元素
<code>\$days{Feb}</code> " "	散列 <code>%days</code> 中“Feb”对应的值

需要说明，对`$days`、`@days`和`%days`可以使用相同的名字（`days`），Perl不会搞错。

一些特殊情况下（你可能不会遇到这些情况），还有另外一些更有意思的标量项。表2-2给出了这些标量项。

表2-2：标量项的语法

构造	含义
<code>\${days}</code>	与 <code>\$days</code> 相同，不过在字母数字字符前没有二义性
<code>\$Dog::days</code>	Dog包中的另一个不同的 <code>\$days</code> 变量
<code>\$#days</code>	数组 <code>@days</code> 的最后一个索引
<code>\$days-&gt;[28]</code>	引用 <code>\$days</code> 指向的数组中的第29个元素
<code>\$days[0][2]</code>	多维数组
<code>\$days{2000}{" Feb"}</code>	多维散列
<code>\$days{2000, " Feb"}</code>	模拟多维散列

整个数组（或数组和散列的片段）命名要以印记`@`开头，这就像单词“these”或“those”。表2-3给出了这个语法。

表2-3：列表项的语法

构造	含义
<code>@days</code>	包含（ <code>\$days[0]</code> ， <code>\$days[1]</code> ，... <code>\$days[N]</code> ）的数组
<code>@days[3, 4, 5]</code>	包含（ <code>\$days[3]</code> ， <code>\$days[4]</code> ， <code>\$days[5]</code> ）的数组片段
<code>@days[3..5]</code>	包含（ <code>\$days[3]</code> ， <code>\$days[4]</code> ， <code>\$days[5]</code> ）的数组片段
<code>@days{Jan, "Feb"}</code>	包含（ <code>\$days{“Jan”}</code> ， <code>\$days{“Feb”}</code> ）的散列片段

整个散列命名以`%`开头，如表2-4所示。

表2-4：散列项的语法

构造	含义
<code>%days</code>	（Jan => 31, Feb => \$leap ? 29 : 28, ...）



这些构造都可以作为左值，它们指定了一个位置，可以在其中赋值。对于数组、散列和数组或散列片段，左值会提供多个可赋值的位置，所以可以一次为它们赋多个值：

```
@days = 1 .. 7;
```

## 名字

我们已经讨论了可以在变量中存储值，不过变量本身（它们的名字和相关定义）也需要存储在某个地方。概念上讲，这些地方称为命名空间（namespaces）。Perl提供了两类命名空间，通常称为符号表（symbol tables）和词法作用域（lexical scopes）<sup>注5</sup>。可以有任意多个符号表或词法作用域，不过你定义的每一个名字都只能存储在某一个符号表或词法作用域中（而不能同时存储在多个命名空间中）。在后面的学习中，我们还会解释这两类命名空间。现在只需要知道，符号表是全局散列，其中包含全局变量（包括其他符号表的散列）的符号表记录。与之相反，词法作用域是未命名（unnamed）的便签簿（scratchpad），它们不在任何符号表中，而与程序中的一个代码块关联。词法作用域包含只有这个代码块能看到的变量[这就是作用域（scope）一词的含义。这里的词法（lexical）则表示“与文本有关”，这可不是通常词典里的意思。请不要因此责怪我们]。

在任何给定的命名空间中（不论是全局还是词法作用域），每个变量类型都有自己的子命名空间，这由印记确定。你可以为一个标量变量、数组或散列（甚至文件句柄、子例程名、标签或你的宠物骆驼）使用相同的名字，而不用担心冲突。这意味着，\$foo和@foo是两个不同的变量。结合前面的规则，这还说明\$foo[1]是@foo的一个元素，与标量变量\$foo完全无关。看起来可能有些怪异，不过没关系，因为它确实很怪异<sup>注6</sup>。

子例程命名可以用&开头，不过调用子例程时，这个印记是可选的。一般不认为子例程是左值，不过你可以请Perl允许你从子例程返回一个左值，然后为它赋值，这样看起来就像是对子例程赋值一样。

有时你只想为“所有名叫foo的东西”（不论它的印记是什么）起一个名字。为此，符号表记录可以有一个\*作为前缀，这里的星号（\*）表示所有其他印记。这些称为类型团（typeglobs），它们有很多用途。类型团也可以作为左值。Perl从一个符号表向另一个符号表导入符号时就是在对类型团赋值。后面还会介绍更多有关内容。

与大多数计算机语言类似，Perl有一组保留字，它们将识别为特殊的关键字。不过，由于变量名总是以一个印记开头，所以保留字实际上不会与变量名冲突。不过，某些名字没有

注5：讨论Perl的特定实现时，我们也称之为包（packages）和pads。尽管前面的名字（“符号表”和“词法作用域”）比较长，但它们是通用的行业术语，所以我们还是采用这种通用说法。抱歉。

注6：实际上，这确实太怪异了，所以我们决定在Perl 6中采用另外一种做法，不过那种做法又在其他方面很怪异。



印记，如标签和文件句柄。对于这些名字，确定要（稍稍）注意不要与保留字冲突。由于大多数保留字都用全小写，所以建议你选择包含大写字符的标签和文件句柄名。例如，如果写为`open(LOG, logfile)`而不是让你后悔的`open(log, "logfile")`，就不会让Perl误以为你所说的是内置的`log`操作符（内置`log`操作符会完成对数运算，而不是你想表示的树干“log”）。使用大写的文件句柄名还可以提高可读性<sup>注7</sup>，避免与我们将来可能增加的保留字冲突。出于类似的原因，对于用户自定义的模块，通常使用首字母大写的模块名，使它们看上去与内置模块（称为`pragmas`）不同，因为内置模块名都是小写。谈到面向对象编程时，你会注意到，出于同样的原因，类名总是首字母大写。

从上一段你应该可以得出一个结论：大小写在标识符中非常重要，`FOO`、`Foo`和`foo`在Perl中是完全不同的名字。标识符以字母或下划线开头，可以是任意长度（这里的“任意”的范围是1到251，包括1和251），其中可以包含字母、数字和下划线。如果使用`use utf8`声明你的源代码是Unicode，这些规则需要稍稍做点调整：标识符必须以连接符号（如下划线）或任何有Unicode `XID_Start` (XIDS)属性的字符开头，后面可以是任何有`XID_Continue` (XIDC)属性的字符。这样一来，你的标识符可以选择100 000多个不同的字符<sup>注8</sup>，包括象形文字，这会当作字母，不过我们不建议使用象形文字，除非你能正常阅读<sup>注9</sup>。参见第6章。

严格地讲，印记后面的名字不一定是标识符。可以以数字开头，在这种情况下，后面只能包含更多数字，如`$123`。如果不是以字母、数字或连接符号开头，而是以任何其他符号开头，这样的名字（通常）只包含这一个字符（如`$?`或`$$`），这在Perl中往往有预定义的含义。例如，与UNIX shell中一样，`$$`是当前进程ID，`$?`是上一个子进程的退出状态。

Perl对于内部变量名还有一套可扩展语法。形如`${^NAME}`的变量都是为Perl保留的特殊变量，只能由Perl使用。所有这些不作为标识符的名字都必须放在主符号表中。可以参见第25章，其中给出了一些例子。往往会认为标识符和名字是一样的，不过，说到“名字”（*name*），通常是指一个完全限定名（*fully qualified name*）；也就是说，这个名字会指出它在哪个符号表中。这种名字通常由一个标识符序列构成，各标识符之间用`:: token`分隔：

```
$Santa::Helper::Reindeer::Rudolph::nose
```

这有些类似于路径名中的目录和文件名：

---

注7： Perl的设计原则之一就是要让不同的东西看起来不同。有些语言则与此完全不同，它们总想让不同的东西看起来相同，这会降低可读性。

注8： 写这本书时，这是指Unicode v6.0。

注9： 对于v5.14，Perl并没有规范化变量名，所以即使名字看起来相同，但如果一个有组合字符，而另一个只有非组合字符，二者实际上并不相同。

在Perl中，完全限定名中的所有前导标识符都是嵌套符号表的名字，最后一个标识符才是嵌套最深的符号表中的变量名。例如，在上面的变量中，符号表名为Santa::Helper::Reindeer::Rudolph::，这个符号表中的具体变量名为\$nose（当然，这个变量的值为“red”）。

Perl中的符号表也称为包（package），所以这也通常也称为包变量。名义上，包变量对于它所在的包是私有的，不过由于包本身是全局的，所以这些包变量也是全局的。也就是说，任何人都可以指定包来访问其中的包变量，这是很普遍的。例如，如果程序提到\$Dog::bert，就是在请求Dog::包中的\$bert变量。这与\$Cat::bert是完全不同的变量。参见第10章。

关联到词法作用域的变量不放在包中，所以词法作用域变量的名字不包含::序列（词法作用域变量用my、our或state声明来声明）。

## 名字查找

现在要回答的问题是，名字里有什么？如果你只说了\$bert，Perl怎么知道你是什么意思？很高兴你能问这个问题。Perl解析器在理解上下文中的一个未限定名时会使用以下规则：

1. 首先，Perl会查看当前代码块，确定这个代码块是否已声明该变量（使用my、our或state声明）。这些内容可以参见第27章，另外可以参见第4章中的“作用域声明”一节。如果有一个my或state声明，则说明这个变量为词法作用域变量，它不在任何包中，只存在于这个词法作用域中（也就是说，在这个代码块的便签簿中）。由于词法作用域是未命名的，任何人都无法在该程序块之外访问这个变量<sup>注10</sup>。
2. 如果没有这样的声明，Perl会查找包围这个代码块的上一层代码块（外围代码块），在这个更大的代码块中查找是否有这个词法作用域变量。同样的，如果Perl找到了这样一个变量声明，说明该变量只属于从该声明到它所在代码块末尾的这个词法作用域（包括所有嵌套的代码块，如第1步查找的那个代码块）。如果Perl没有找到这样一个声明，它会重复步骤2，直到再无外围代码块。
3. 如果Perl发现再无外围代码块，则它会在整个编译单元中检查声明，就好像这个编译单元是一个代码块一样。编译单元（compilation unit）就是整个当前文件，或者是一

---

注10：如果使用一个our声明而不是my或state声明，这只会为包变量声明一个词法作用域别名（外号），而不是像my或state那样声明一个真正的词法作用域变量。外部代码仍然可以通过变量的包访问具体变量，不过在所有其他方面，our声明的表现与my声明是一样的。如果你想用strict pragma限制全局变量的使用，这会很方便（只有声明了use v5.14;时才会默认启用这个特性，有关细节请参见第5章的strict pragma）。不过，如果不需要全局变量，最好还是使用my或state声明。



个`eval STRING`操作符当前编译的字符串。如果编译单元是一个文件，这就是可能的最大词法作用域，Perl不会再继续寻找词法作用域变量，所以将进入步骤4。不过，如果编译单元是一个字符串，这就比较有意思了。运行时编译为Perl代码的字符串会假装这是一个代码块，处于运行`eval STRING`的词法作用域中（尽管这个词法作用域的实际边界是包含代码的字符串边界，而不是真正的大括号）。所以，如果Perl没能在该字符串的词法作用域中找到这个变量，我们就假装`eval STRING`是一个块，然后再回到步骤2，只不过这一次是从`eval STRING`操作符的词法作用域开始，而不是从字符串内部的词法作用域开始。

4. 如果到这一步，说明Perl没有找到该变量的任何声明（`my`或`our`）。现在Perl会放弃寻找词法作用域变量，而假设这个变量是一个包变量。如果启用了`strict pragma`，现在就会得到一个错误，除非这个变量是Perl的一个预定义变量，或者已经导入到当前包中。这是因为，这个`pragma`不允许使用非限定全局名。不过，词法作用域还没有处理完。Perl会像步骤1到步骤3中一样在词法作用域中完成同样的搜索，只不过这一次它会搜索包声明而不是变量声明。如果确实找到这样一个包声明，它就会知道当前代码是为当前这个包编译的，并将声明的包名附加到变量前面。
5. 如果外围词法作用域中都没有找到包声明，Perl会在未命名的顶级包中查找变量名，如果没有指定名字标记，就会有一个名字`main`。所以，如果没有任何声明，`$bert`的含义就与`$::bert`相同，也等同于`$main::bert`（不过，由于`main`只是顶级未命名包中的另一个包，所以这个变量也等同于`$::main::bert`、`$main::main::bert`、`$::main::main::bert`等。尽管这一点是事实，但并没有什么用。请参见第10章的“符号表”一节）。

这些搜索规则还有一些不太明显的隐含内容，下面将逐一说明。

- 因为文件是可能的最大词法作用域，对于词法作用域变量来说，在声明它的文件之外该变量是不可见的。文件作用域不嵌套。
- 任何Perl代码至少在一个词法作用域中编译，而且只在一个包作用域中。当然，它肯定有一个词法作用域，即文件本身。其他词法作用域则由各个外围代码块提供。所有Perl代码只在一个包作用域中编译，尽管相应的包声明在词法作用域中，但包本身不受词法作用域限制。也就是说，包是全局的。
- 因此可能会在多个词法作用域中搜索未限定变量名，但是只在一个包作用域中搜索（也就是当前起作用的那个包，这由词法确定）。
- 一个变量名可能只与一个作用域关联。尽管程序中任何地方至少有两个不同的作用域（词法作用域和包作用域），但变量只能存在于其中一个作用域中。
- 因此未限定变量名可以解析为一个存储位置，可能是声明该变量的第一个外围词法作用域，也可以是当前包，但不能二者同时。一旦解析到这个存储位置，搜索就停



止，这将有效地隐藏所有其他可能的存储位置（如果搜索继续，则有可能找到那些存储位置）。

- 一般变量名的位置完全可以在编译时确定。

现在你知道了Perl编译器如何处理名字，但有时还会遇到另一个问题：编译时你并不知道所要的那个名字。有时你想间接地为变量命名，我们把这个问题称为间接（indirection）。为此Perl提供了一种机制：可以把一个字母数字变量名替换为包含有一个表达式的代码块，这个表达式会返回实际数据的一个引用（reference）。例如，可以不这么说：

```
$bert
```

而是这么说：

```
${ some_expression() }
```

如果`some_expression()`函数返回变量`$bert`（或者甚至是字符串“bert”）的一个引用，其效果与前面的`$bert`是一样的。另一方面，如果这个函数返回`$ernie`的一个引用，就会得到这个变量。这里显示的语法是最一般的（也是最不好懂的）间接形式，不过我们还会在第8章介绍另外一些简便的间接语法。

## 标量变量

不论直接还是间接命名，另外不论在一个变量中、数组元素中或者只是一个临时值，标量总包含单个值。这个值可以是一个数字、一个字符串或者另外一个数据的引用。或者，甚至可以根本没有值，在这种情况下，我们称这个标量是未定义的（undefined）。尽管我们可能说标量“包含”一个数字或一个字符串，但实际上标量是无类型的：没有必要声明标量是整数、浮点数、字符串或是任何其他类型。

以后的Perl版本可能允许你插入`int`、`num`和`str`类型声明。这并不是强制强类型，而只是为优化工具提供一些暗示，否则优化工具自己可能无法确定这些信息。一些CPAN模块已经可以做到这一点。

Perl将字符串存储为字符序列，对于长度或内容没有任何限制。对人来说，你没有必要提前确定字符串会有多长，另外可以在字符串中包含任意字符，包括null字节。Perl尽可能将数字存储为有符号（或无符号）整数，如果不能存储为整数，则采用机器的内部格式存储为双精度浮点值。浮点值不是无限精确的。要记住这一点，这很重要，因为像`(10/3 == 1/3*10)`之类的比较会莫名其妙地失败。

不过，通过使用`bigint`、`bigrat`和`bignum` pragmas，可以改用Perl的常用数字记法。它们可以提供任意精度的整数、有理数（分数）和浮点数。这会更符合你的期望：

```
% perl -E 'say 10/3 == 1/3*10 ? "Yes" : "No" '
No

% perl -Mbigrat -E 'say 10/3 == 1/3*10 ? "Yes" : "No" '
Yes

% perl -E 'say 4/3 * 5/12'
0.5555555555555555

% perl -Mbigrat -E 'say 4/3 * 5/12'
5/9
```

在程序中（而不是在命令行中），要使用声明`use bigint`、`use bigrat`和`use bignum`来得到这些更有趣的数字：

```
use v5.14;
use bigrat;
say 1/3 * 6/5 * 5/4; # 打印"1/2"
```

Perl会根据需要在不同子类型之间完成转换，所以可以把一个数字当成字符串，或者把一个字符串当作数字处理，Perl绝不会失误。要把字符串转换为数字，Perl会在内部使用一个转换函数，类似于C库中的`atof(3)`函数。要把数字转换为字符串，它会完成类似`sprintf(3)`的处理（在大多数机器上会采用`%.14g`格式）。要将非数值字符串（如`foo`）不适当地转换为一个数字，这会得到数值0；如果你启用了警告，这种不适当的转换就会触发警告，否则不会有任何表示。关于如何检测一个字符串中包含何种数据，第5章给出了一些例子。

尽管字符串和数字几乎在所有情况下都可以互换，不过引用稍有些差别。引用是强类型的指针，而且其类型不可强制转换，它提供了内置的引用计数和析构调用。也就是说，可以用引用来创建复杂的数据类型，包括用户自定义的对象。不过尽管如此，它们仍是标量，因为无论一个数据结构变得多复杂，你总希望把它当作单个值。

这里提到“不可强制转换类型”，举例来说，这是指不能把一个数组的引用转换为一个散列的引用。引用不能转换为其他指针类型。不过，如果把引用用作一个数字或字符串，你会得到一个数值或字符串值，这可以保证引用的唯一性，尽管从实际引用复制值时会丢掉这个值的“引用性”。可以比较这些值，或者抽取这些值的类型。不过对这些值无法做太多其他处理，因为没有办法把数字或字符串再转换回引用。通常这不成问题，因为Perl不要求（甚至不允许）完成指针运算。有关引用的更多内容请参见第8章。

## 数值直接量

数值直接量可以用我们惯用<sup>注11</sup>的多种浮点数或整数格式指定：

---

注11：也就是说，这些是UNIX文化中惯用的格式。如果你来自另一个不同的文化，欢迎加入我们！



```

my $x = 12345;           # 整数
my $x = 12345.67;        # 浮点数
my $x = 6.02e23;         # 科学计数学
my $x = 4_294_967_296;   # 加下划线以便辨认
my $x = 0377;            # 八进制
my $x = 0xffff;          # 十六进制
my $x = 0b1100_0000;     # 二进制

```

由于Perl使用逗号作为列表分隔符，不能将它用作大数字中的千位分隔符。不过，Perl允许用下划线字符作为千位分隔符。下划线只能用于程序中指定的直接量数字，而不能用于作为数字的字符串或者从别处读取的数据。类似的，十六进制前面的0x、二进制前面的0b，以及八进制前面的0只适用于直接量。字符串自动转换到数字时，无法识别这些前缀，你必须用oct函数完成显式转换<sup>注12</sup>。如果前面指定0x或0b，这也可以用来转换十六进制和二进制数字。

## 字符串直接量

字符串直接量通常由单引号或双引号引起。它们类似于UNIX shell的引用（quotes）：双引号字符串直接量支持反斜线和变量内插，不过单引号字符串不允许（除\ '和\\以外，所以可以在单引号字符串中嵌入单引号和反斜线）。如果想嵌入任何其他反斜线序列（如换行符\n），必须使用双引号形式（反斜线序列也称为转义序列（escape sequences），因为你要临时“转义”字符的正常解释）。

单引号字符串必须用一个空格与前面的单词分隔开，因为单引号在标识符中是合法的字符（尽管有些怪异）。现在已经把它替换为更容易区分的::序列。这说明，\$main'var 和 \$main::var是同一个东西，不过后者对于人和程序来说更易读。

双引号字符串支持各种形式的字符内插，如表2-5所列。其中很多形式对于使用其他语言的程序员来说应该已经很熟悉了。

表2-5：反斜线字符转义

代码	含义
\n	换行（通常写作LF）
\r	回车（通常写作CR）
\t	水平制表符
\f	换页

注12：有时人们认为Perl应该为他们转换所有数据。不过这个世界上有太多有前导0的十进制数字，所以Perl无法自动完成转换。例如，位于马萨诸塞州剑桥的O'Reilly Media办公室邮政编码为02140。如果你的邮件标签程序把02140转换为十进制数1120，会让邮递员不知所措。



表2-5：反斜线字符转义（续）

代码	含义
<code>\b</code>	退格
<code>\a</code>	警报（响铃）
<code>\e</code>	ESC字符
<code>\033</code>	ESC的八进制形式
<code>\o{33}</code>	也是ESC的八进制形式
<code>\x7f</code>	DEL的十六进制形式
<code>\x{263a}</code>	字符数字0x263A
<code>\N{LATIN SMALL LETTER E WITH ACUTE}</code>	命名字符 LATIN SMALL LETTER E WITH ACUTE, “é”，Unicode中相应的码点为0xE9
<code>\N{ U+E9 }</code>	这也是字符数字0xE9
<code>\cC</code>	Ctrl-C

`\N{NAME}`记法只能结合`charnames pragma`（将在第29章介绍）使用，允许你象征地指定字符名，如`\N{GREEK SMALL LETTER SIGMA}`、`\N{greek:Sigma}`或`\N{sigma}`，这要看你如何调用`pragma`。`\N{U+HEXDIGITS}`记法不要求声明`charnames pragma`，完全可以保证采用这种记法的字符串或正则表达式一定使用Unicode语义，参见第6章。

还有一些转义序列可以修改后面字符的大小写或“元性”，见表2-6。

表2-6：转换转义

代码	含义
<code>\u</code>	强制下一个字符为标题形式（titlecase） <sup>a</sup>
<code>\l</code>	强制下一个字符为小写
<code>\U</code>	强制后面的所有字符为大写，在 <code>\E</code> 处结束
<code>\L</code>	强制后面直到 <code>\E</code> 的所有字符为小写，在 <code>\E</code> 处结束
<code>\F</code>	强制后面直到 <code>\E</code> 的所有字符为foldcase形式，在 <code>\E</code> 处结束 <sup>b</sup>
<code>\Q</code>	为后面所有非字母数字字符加反斜线，在 <code>\E</code> 处结束
<code>\E</code>	结束 <code>\U</code> 、 <code>\L</code> 、 <code>\F</code> 或 <code>\Q</code>

a: titlecase是一种Unicode形式，非常类似于大写（uppercase），参见第6章。

b: `\F`是v5.16新增的。foldcase是一种特殊的形式，用来完成不区分大小写的比较。参见第5章和第6章。

你也可以在字符串中直接嵌入换行符。也就是说，字符串可以换行（可以在不同的行开始

和结束)。这往往很有用，不过这也说明如果你忘记加结尾的引号，将不会报告这个错误，直到Perl看到另一个包含引号字符的代码行时才会发现错误，而这行代码可能在脚本很后面的位置。幸运的是，如果在同一行上，这会立即导致一个语法错误，Perl很聪明，它会警告你字符串可能没有正常结束。

除了上面所列的反斜线转义之外，双引号字符串还支持标量和列表值的变量内插（variable interpolation）。这说明，你可以把某些变量的值直接插入到一个字符串直接量中。这实际上是实现字符串连接的一种简便方式<sup>注13</sup>。可以对标量变量、整个数组（但不包括散列）、数组或散列中的单个元素，或者数组或散列的片段（多个下标）使用变量内插。除此之外，其他变量都不允许内插。换句话说，只能对以\$或@开头的表达式完成内插，因为字符串解析器会查找这两个字符（连同反斜线）。在字符串中，如果@不是数组或片段标识符的一部分，并且后面跟有一个字母数字字符，它就必须用一个反斜线转义（\@），否则就会导致一个编译错误。虽然用一个%指定的完整散列不能内插到字符串中，但是单个散列值或散列片段是可以的，因为它们分别以\$和@开头。

下面的代码段会打印出“The price is \$100.”：

```
my $Price = '$100';           # 未内插
print "The price is $Price.\n"; # 内插
```

与一些shell相似，可以在标识符外加大括号，以便区分后面的字母数字：“How \${verb} able!”。这种大括号内的标识符会强制为一个字符串，类似于散列下标中的标识符。例如：

```
$days{"Feb"}
```

可以写作：

```
$days{Feb}
```

这里假设有引号。下标中更复杂的内容要解释为一个表达式，这些需要放在引号中：

```
$days{'February 29th'} # 可以
$days{"February 29th"} # 也可以。""不用内插
$days{ February 29th } # 不可以，会产生解析错误
```

特别地，片段中一定要加引号，如下：

```
@days{'Jan','Feb'} # 可以
@days{"Jan","Feb"} # 也可以
@days{ Jan, Feb }   # 可能出错（如果有use strict声明就会出错）
```

---

注13：如果启用了警告，Perl可能会报告在使用连接或结合操作时字符串中插入了未定义的值，尽管你实际上并没有使用这些操作符。不过编译器会为你创建。

除了内插数组和散列变量的下标外，不存在其他多层内插。出乎shell程序员预料，双引号中的反引号不会完成内插，另外单引号在双引号字符串中使用时也不会停止变量的计算。在Perl中，内插是一个非常强大的概念，但也受到严格控制。内插只能在双引号字符串中进行，另外下一节将要介绍的另外一些“类双引号”操作中也可以完成内插：

```
print "\n";      # 可以，打印一个换行符
print \n;        # 不正常，没有提供可内插的上下文
```

## 选择你自己的引号

尽管我们通常认为引号是直接量值，实际上在Perl中它们更类似于操作符，提供了多种内插和模式匹配功能。Perl为这些功能提供了常用的引号字符，另外还提供了一种更一般的方法，允许你为这些功能选择自己的引号字符。在表2-7中，/可以替换为任何非字母数字且非空白符的定界符（换行符和空格字符不允许作为定界符，尽管很早以前的Perl版本曾经允许这样）。

表2-7：引号构造

常用	通用	含义	内插
' '	q//	直接量字符串	不支持
" "	qq//	直接量字符串	支持
` `	qx//	命令执行	支持
()	qw//	单词列表	不支持
//	m//	模式匹配	支持
s///	s///	模式替换	支持
tr///	y///	字符转换	不支持
" "	qr//	正则表达式	支持

其中一些相当于“语法糖”，可以帮你避免在引号字符串中放入太多的反斜线，特别是模式匹配中，其中正常的斜线和反斜线往往会“纠缠”在一起。

如果选择单个引号作为定界符，即使采用正常情况下支持内插的形式，也不会完成变量内插。如果开始定界符是一个开始小括号、中括号、大括号或尖括号，结束定界符就必须是相应的结束字符（内嵌的定界符必须成对匹配）。例如：

```
my $single = q!I said, "You said, 'She said it.'!";

my $double = qq(Can't we get some "good" $variable?);

my $chunk_of_code = q {
    if ($condition) {
        print "Gotcha!";
    }
}
```



```
    }  
};
```

上面这个例子展示了可以在引号指示符和第一个定界字符之间使用空白符。对于类似`s///`和`tr///`等两元素的构造，如果第一对引号是一对定界字符，第二部分会有其自己的开始引号字符。实际上，第二对不必与第一对相同。所以还可以写为`s<foo>(bar)`或`tr(a-f)[A-F]`。由于两个内部引号字符之间允许有空白符，所以甚至可以把最后一个例子写为：

```
tr (a-f)  
[A-F];
```

不过，如果用`#`作为引号字符，则不允许有空白符。`q#foo#`解析为字符串`'foo'`，而`q #foo#`会解析为引号操作符后面跟有一个注释。它的定界符将从下一行选取。也可以在两元素构造中间放入注释，所以可以写为：

```
s {foo}      # 将foo  
{bar};      # 替换为bar  
  
tr [a-f]     # 将小写十六进制字符  
[A-F];      # 替换为大写十六进制字符
```

## 或者完全省略引号

如果一个名字的文法中没有其他内插，可以把它当作一个引号字符串。这称为裸字（barewords）<sup>注14</sup>。类似于文件句柄和标签，如果裸字完全由小写的ASCII字母组成，这就存在与将来的保留字发生冲突的风险。如果启用了警告，Perl会警告你当心这些裸字。例如：

```
my @days = (Mon,Tue,Wed,Thu,Fri);  
print STDOUT hello, " ", world, "\n";
```

将数组`@days`设置为短格式星期名，并在`STDOUT`上打印“hello world”，然后是一个换行符。如果省略这个文件句柄，Perl会把`hello`解释为一个文件句柄，这就会导致一个语法错误。因为这样很容易出错，所以有些人可能希望干脆不要使用裸字。前面列出的引号操作符提供了很多方便的形式，包括`qw//`“引号字”构造，它能很好地引起一个用空格分隔的单词列表：

```
my @days = qw(Mon Tue Wed Thu Fri);  
print STDOUT "hello world\n";
```

甚至还可以声明裸字是不合法的。如果声明：

---

注14：变量名、文件句柄、标签等都不认为是裸字，因为其前导token或后续token（或者二者都有）会为它们指定一个含义。预声明的名字（如子例程）也不是裸字。只有当解析器没有任何暗示时子例程才是裸字。

```
use strict "subs";
```

这样一来，所有裸字都会生成一个编译时错误。这个限制将一直持续到外围作用域的最后。在内部作用域中可以取消这个限制：

```
no strict "subs";
```

声明裸字不合法是个好主意，如果有

```
use v5.12;
```

或更高版本，Perl会自动为你打开所有约束。

注意在如下的构造中，

```
"${verb}able"  
$days{Feb}
```

并不认为裸标识符是裸字，因为有明确的规则支持裸标识符，而不是考虑到“文法中没有其他内插”。

如果未加引号的名字后面有一个双冒号，如`main::`或`Dog::`，这总会当作包名。Perl在编译时将可能的裸字`Camel::`转换为字符串“`Camel`”，所以这种用法是完全可以的，不应受到责难。

## 内插数组值

数组变量内插到双引号字符串中时，要把数组的所有元素用`$`变量<sup>注15</sup>（其中默认包含一个空格中指定的分隔符）连接起来。下面几个语句是等价的：

```
my $temp = join( $", @ARGV );  
print $temp;  
  
print "@ARGV";
```

在搜索模式中（也要完成双引号内插），存在一种让人遗憾的二义性：`/${foo}[bar]/`解释为`/${foo}[bar]/`（其中`[bar]`是表示正则表达式的一个字符类）？还是解释为`/${foo[bar]}/`（其中`[bar]`是数组`@foo`的下标）？如果`@foo`不存在，显然这是一个字符类。不过，如果`@foo`存在，Perl也会适当地猜测`[bar]`，而且往往不会猜错<sup>注16</sup>。如果确实猜错了，或者如果你有特别的想法，可以用大括号强制完成正确的内插（如前所示）。即使你很谨慎也不妨这么做，这不是一个坏主意。

---

注15：如果使用Perl提供的English模块，则是`$LIST_SEPARATOR`。

注16：猜测模式含义的方法很枯燥，这里就不详细介绍了，不过基本说来就是字符类（`a-z`、`\w`和初始`^`）或是表达式（变量或保留字）。

## “Here” 文档

面向行的引用 (quote) 建立在UNIX shell的*here-document*语法基础上。由于定界符是行而不是字符，所以从这个意义上讲它是面向行的。起始定界符是当前行，终止定界符则是一个包含你指定的某个字符串的行。可以在一个<<后面指定用来终止引用内容的字符串，这样一来，当前行直到（但不包括）终止行之间的所有行都将属于这个字符串。终止符字符串可以是一个标识符（单词），也可以是一个加引号的文本。如果有引号，引号的类型将确定如何处理这个文本，就像正常的引号串中一样。对于未加引号的标识符，就像在双引号中一样处理。加反斜线的标识符则像在单引号中一样处理（为了与shell语法兼容）。<<和未加引号的标识符之间不允许有空格，不过如果指定了一个加引号的字符串而不是裸标识符，则允许有空白符（如果插入一个空格，将把它当作一个null标识符，这是合法的，但这种做法已经过时，可以匹配第一个空行，请参见下面第一个Hurrah!例子）。终止符字符串必须在终止行上单独出现，不能有引号，两边也不能有多余的空白符。

```
print <<EOF; # 与前一个例子相同
The price is $Price.
EOF
```

```
print <<"EOF"; # 与上面相同，有显式的引号
The price is $Price.
EOF
```

```
print <<'EOF'; # 单引号引用
All things (e.g. a camel's journey through
A needle's eye) are possible, it's true.
But picture how the camel feels, squeezed out
In one long bloody thread, from tail to snout.
-- C.S. Lewis
EOF
```

```
print <<\EOF;      # 另一个单引号引用
I could really use $100 about now.
EOF
```

```
print << x 10;      # 将下一行打印10次
The camels are coming! Hurrah! Hurrah!
```

```
print <<" " x 10;    # 更优先的写法
The camels are coming! Hurrah! Hurrah!
```

```
print <<`EOC`;      # 执行命令
echo hi there
echo lo there
EOC
```

```
print <<"dromedary", <<"camelid"; # 可以叠加
I said bactrian.
dromedary
She said llama.
```



```
camelid
```

```
funkshun(<<"THIS", 23, <<'THAT'); # 在父括号中也没有关系
Here's a line
or two.
THIS
And here's another.
THAT
```

不要忘记，要在末尾加一个分号来结束语句，否则Perl不知道你想要结束语句：

```
print <<"odd"
2345
odd
    + 10000; # 打印12345
```

如果希望你的here文档相对于其余代码缩进，需要手动地从各行删除前导空白符：

```
(my $quote = <<'QUOTE') =~ s/^\s+//gm;
    The Road goes ever on and on,
    down from the door where it began.
QUOTE
```

甚至可以用一个here文档的各行填充一个数组，如下所示：

```
my @sauces = <<End_Lines =~ m/(\S.*\S)/g;
    normal tomato
    spicy tomato
    green chile
    pesto
    white wine
End_Lines
```

## 版本直接量

如果一个直接量以v开头而且后面跟有一个或多个用逗号分隔的十进制整数，这会作为一个版本号：

```
use v5.14; # 打开strict和警告
```

这些原来称为v串（v-strings），不过用这种记法生成字符串值的做法已经过时。现在只能使用这种记法生成版本对象。

## 其他直接量Token

开头和结尾有一个双下划线的标识符都应看作是为Perl保留的，它们有特殊的语法用途。\_\_LINE\_\_和\_\_FILE\_\_就是这样两个特殊的直接量，分别表示程序中的当前行号和文件名。它们只能用作单独的token；不能内插到字符串中。类似地，\_\_PACKAGE\_\_是当前的包名（当前行将要编译到这个包中）。token \_\_END\_\_（或者Control-D或Control-Z字符）可

以用来指示脚本的逻辑结束（在真正的文件末尾之前）。后面的所有文本都将被忽略，但是可以通过DATA文件句柄读取。

`__DATA__` token作用与`__END__` token相似，不过它会在当前包的命名空间打开DATA文件句柄，所以你用`require`包含的文件可以同时分别有自己的DATA文件句柄。有关的更多信息，请参见第25章的DATA。

## 上下文

到目前为止，我们已经了解了很多可以生成标量值的项。不过，在进一步讨论其他项之前，需要谈一谈项的上下文（context）。

### 标量和列表上下文

Perl脚本中调用的每一个操作<sup>注17</sup>会在一个特定的上下文计算和执行，这个操作的行为取决于这个上下文的需求。有两种主要的上下文：标量上下文和列表上下文。例如，对标量变量赋值时，或者对数组或散列的一个标量元素赋值时，就会在标量上下文中计算右边的值：

```
$x      = funkshun(); # 标量上下文
$x[1]   = funkshun(); # 标量上下文
$x{"ray"} = funkshun(); # 标量上下文
```

不过，对数组或散列赋值时，或者对数组或散列的一个片段赋值时，会在一个列表上下文中计算右边（即使片段只选出一个元素）：

```
@x      = funkshun(); # 列表上下文
@x[1]   = funkshun(); # 列表上下文
@x{"ray"} = funkshun(); # 列表上下文
%x      = funkshun(); # 列表上下文
```

为一个标量列表赋值时，也会为右边提供列表上下文，即使列表中只有一个元素：

```
($x,$y,$z) = funkshun(); # 列表上下文
($x)       = funkshun(); # 列表上下文
```

用`my`、`state`或`our`声明变量时，这些规则不会改变，所以有：

```
my $x      = funkshun(); # 标量上下文
my @x      = funkshun(); # 列表上下文
my %x      = funkshun(); # 列表上下文
my ($x)    = funkshun(); # 列表上下文
```

---

注17：这里“操作”的含义很宽松，可以表示操作符，也可以表示一个项。讨论一些函数时这两个概念有些含糊，这些函数解析时像是项，但看起来像一元操作符。

你要了解标量上下文和列表上下文之间的区别，否则可能会遇到麻烦，因为有些操作符（如前面这个虚构的funktshun函数）知道自己所在的上下文，在需要列表的上下文中会返回一个列表，而在希望得到标量的上下文中会返回一个标量值（如果一个操作可以做到这一点，会在该操作的文档中指出）。用计算机术语来讲，这些操作的返回类型是重载的（overloaded）。不过这是一种很简单的重载，只基于单数和复数值之间的差别，而不考虑其他方面。

如果某些操作符能响应上下文，显然它们周围的某个东西要提供这个上下文。我们已经展示了赋值可以为右操作数提供一个上下文，不过这并不奇怪，因为所有操作符都会为它的各个操作数提供某种上下文。你想知道的是，哪个操作符为它的操作数提供了哪个上下文。事实上，可以很容易地区分出哪些操作符能提供列表上下文，因为它们的语法描述中都有LIST。所有其他操作符则提供标量上下文。通常这很直观<sup>注18</sup>。如果有必要，可以使用scalar伪函数为LIST中的一个参数强制标量上下文。Perl没有提供强制列表上下文的方法，因为如果希望有列表上下文，可由某个控制函数的LIST提供。

标量上下文可以进一步划分为字符串上下文、数值上下文和“无关”上下文。与标量上下文和列表上下文的区别不同，操作本身并不知道也不关心到底在哪个标量上下文中。操作只是返回需要的那种标量值，由Perl在字符串上下文中把数字转换为字符串，在数值上下文中将字符串转换为数字。有些标量上下文不关心返回的是字符串、数字或是引用（因此称之为“无关”或“不关心”上下文），所以不会发生任何转换。例如，为另一个变量赋值时就是这种情况。新变量会取原值同样的子类型。

## 布尔上下文

另一个特殊的无关标量上下文称为布尔上下文（Boolean context）。布尔上下文是指，要在这里计算表达式来查看它为true还是false。本书中谈到“true”和“false”时，是指Perl使用的技术定义：如果一个标量值不为空串""或数字0（或其字符串表示"0"），这个标量值就为true。引用总为true，因为它表示一个地址，而地址不会为0。未定义的值（通常称为undef）总是false，因为未定义的值往往是""或0（取决于把它作为一个字符串还是数字）。列表值没有布尔值，因为标量上下文中不会生成列表值！

由于布尔上下文是一个无关上下文，它不会导致任何标量转换，不过当然会为那些关心类型的操作数提供标量上下文。对于这些操作数，标量上下文中生成的标量表示一个合理的布尔值。也就是说，很多操作符可以在列表上下文中生成一个列表，不仅如此，这些操作符还可以在布尔上下文中用于完成true/false测试。例如，在unlink操作符提供的列表上下文中，数组名会生成其值列表：

---

注18：不过，需要说明，LIST的列表上下文可以通过子例程调用向下传播，所以并不总能明显看出一个给定语句会在标量上下文还是列表上下文中计算。程序可以在子例程中使用wantarray函数得出其上下文。



```
unlink @files; # 删除所有文件，忽略错误
```

不过，如果在一个条件语句中使用这个数组（也就是说，在一个布尔上下文中），数组知道它在标量上下文中，所以只返回数组中的元素个数，这个元素个数将解释为true（只要数组中有元素）。所以，假设你希望在各个文件未能适当删除时发出警告，可以写如下循环来实现：

```
while (@files) {  
    my $file = shift(@files);  
    unlink($file) || warn "Can't delete $file: $!";  
}
```

在这里，@files在while语句提供的布尔上下文中计算，所以Perl会计算数组本身，查看这是一个“真数组”还是一个“假数组”。只要其中包含有文件名，这就是一个真数组，不过一旦取出最后一个文件名，它就会变成假数组。需要说明，我们前面所说的仍然成立。尽管数组包含（且可以生成）一个列表值，但在标量上下文中并不会计算列表值。我们会告诉数组它是一个标量，并让它自己考虑如何返回结果。

不要为此试图使用defined @files。这是不行的，因为defined函数会检查一个标量是否等于undef，但是数组不是标量。只要完成这个简单的布尔测试就可以了。

## Void上下文

还有一种特殊的标量上下文，称为void上下文（void context）。这个上下文并不关心返回值的类型是什么，它甚至不希望得到返回值。从函数工作的角度来看，这与普通的标量上下文没有不同。不过，假如启用了警告，如果你使用了一个不希望有返回值而且没有副作用的表达式（如在没有返回值的语句中），Perl编译器就会发出警告。例如，如果将一个字符串用作一条语句：

```
"Camel Lot";
```

则你可能会得到类似下面的警告：

```
Useless use of a constant in void context in myprog line 123;
```

## 内插上下文

之前提到过，双引号直接量字符串可以完成反斜线内插和变量内插，不过内插上下文[通常也称为“双引号上下文”，因为没人拼得出“interpolative”（内插）]不仅仅应用于双引号字符串。还有另外一些类双引号构造，包括通用的反引号操作符qx//；模式匹配操作符m//；替换操作符s///和引用表达式操作符qr//。替换操作符在完成模式匹配之前在左边完成内插，然后每次左边匹配时再在右边完成内插。

内插上下文只在引号（或类似于引号的构造）内部出现，所以相比标量和列表上下文，把它也称为一个“上下文”可能有些不太公平（也可能是有道理的）。

## 列表值和数组

既然已经讨论了上下文，下面再来讨论列表直接量及其在上下文中有何表现。你已经见过一些列表直接量。列表直接量表示为一系列由逗号分隔的单个值（如果有优先级要求，还需要用小括号把列表括起来）。因为使用多余的小括号（几乎）没有坏处，所以列表值的语法图通常表示如下：

(LIST)

之前我们说过，语法描述中的`LIST`表示要为其参数提供列表上下文。不过，对于这个规则，单独的列表直接量本身是一个例外，因为只有当整个列表在列表上下文中时，它才会为其参数提供列表上下文。列表上下文中，列表直接量的值就是其参数值（按指定的顺序）。如果作为表达式中的一项，列表直接量只是把一系列临时值压入到Perl的堆栈中，然后再由希望得到这个列表的操作符从堆栈中收集这些值。不过，在标量上下文中，列表直接量表现得并不像一个`LIST`，因为它不会为它的值提供列表上下文。相反，只是在标量上下文中计算它的各个参数，并返回最后一个元素的值。这是因为，它实际上就是伪装的C逗号操作符，这是一个二元操作符，总会丢掉左边的值，而返回右边的值。根据我们前面讨论的，逗号操作符的左边确实会提供上下文。因为逗号操作符是左结合操作符，如果有一系列用逗号分隔的值，最终总会得到最后一个值，因为最后一个逗号会丢掉之前逗号生成的所有值。下面将这二者做个比较，以下列表赋值：

```
@stuff = ("one", "two", "three");
```

会把整个列表值赋给数组`@stuff`，而标量赋值：

```
$stuff = ("one", "two", "three");
```

只把值“three”赋给变量`$stuff`。类似于前面提到的`@files`数组，逗号操作符知道它在标量上下文中还是在列表上下文中，并相应地选择行为。

有必要重申一句，列表值与数组并不相同。真正的数组变量知道它的上下文，在列表上下文中它会像一个列表直接量一样返回其内部的值列表。而在标量上下文中，它只返回数组的长度。以下代码将把`$stuff`赋值为3：

```
@stuff = ("one", "two", "three");  
$stuff = @stuff;
```

如果希望将它赋值为“three”，你可能会做出错误的推断，认为Perl会使用逗号操作符规则丢掉`@stuff`放在堆栈上的其他临时值，而只留下一个值。不过它并不是这么做的。`@stuff`数组根本不会把它的值放在堆栈中。实际上，它不会在堆栈中放入它的任何元素



值。它只放入了一个值，即数组的长度，因为它知道这是在标量上下文中。标量上下文中所有项或操作符都不会在堆栈中放入一个列表。相反，只会在堆栈中放入一个标量，这往往并非本该在列表上下文中返回的列表中最后一个值。这是因为，最后一个值不一定是标量上下文中最有用的值。明白了吗？如果还不明白，最好把这一段重读几遍，因为这个内容确实很重要。

再来看提供列表上下文的“真LIST”。到目前为止，我们都假设列表直接量就是直接量列表。不过，正如字符串直接量可以内插其他子串一样，列表直接量也可以内插其他子列表。只要表达式能返回值，都可以用在列表中。以这种方式使用的值可以是标量值，也可以是列表值，不过它们都将成为新列表值的一部分，因为LIST会自动地完成子列表内插。也就是说，计算LIST时，在列表上下文中计算该列表的每一个元素，返回的列表值再“展平”放在LIST中，就好像每个单个元素都是LIST的一个成员。因此，这些数组在LIST中会失去它们的身份<sup>注19</sup>。因此列表：

```
(@stuff,@nonsense,funkshun())
```

将包含@stuff的元素，后面是@nonsense的元素，再后面是在列表上下文中调用子例程&funkshun所返回的值。需要说明，其中某个数组（或全部）可能会内插一个空列表，在这种情况下，会表现得好像在这个位置没有内插任何数组或函数。空列表本身由直接量()表示。与空数组一样，这会作为一个空列表完成内插，因此实际上会被忽略，将空列表内插到另一个列表中时，不会有任何作用。因此，(()),(),()就等价于()。

作为这个规则的一个推论，你可以在任何列表值的末尾加一个可选的逗号。这样可以方便以后回来在最后一个元素后面增加更多元素：

```
@releases = (  
    "alpha",  
    "beta",  
    "gamma",  
);
```

或者，也可以完全不要逗号：指定直接量列表的另一种方法是利用我们前面提到过的qw（引号字）语法。这个构造等价于用空白符分隔单引号字符串。例如：

```
@froots = qw(  
    apple    banana    carambola  
    coconut  guava      kumquat  
    mandarin nectarine  peach  
    pear     persimmon  plum  
);
```

---

注19：有些人可能觉得这是个问题，不过实际上没有问题。如果你不希望数组失去自己的身份，完全可以在列表中内插数组的一个引用，参见第8章。



注意，这里的括号要作为引号字符，而不是常规的小括号。也可以选择尖括号或大括号，或者斜线。不过小括号就可以。

列表值也可以像正常的数组一样有下标。必须把列表放在小括号（真正的小括号）中，以避免二义性。尽管常用这种方式从一个列表获取单个值，但这实际上是列表的一个片段，所以语法为：

```
(LIST)[LIST]
```

举例：

```
# 开始返回列表值
$modification_time = (stat($file))[9];

# 这里有语法错误
$modification_time = stat($file)[9]; # 唉呀，忘加小括号了

# 找到一个十六进制数
$hexdigit = ("a","b","c","d","e","f")[$digit-10];

# 一个“反向逗号操作符”
return (pop(@foo),pop(@foo))[0];

# 得到多个值作为一个片段
($day, $month, $year) = (localtime)[3,4,5];
```

## 列表赋值

只有当列表中各元素本身能赋值时，才能为列表赋值：

```
($a, $b, $c) = (1, 2, 3);

($map{red}, $map{green}, $map{blue}) = (0xff0000, 0x00ff00, 0x0000ff);
```

在列表中，元素可以赋值为undef。如果一个函数要舍弃某些返回值，这会很有用：

```
($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

甚至可以在my声明中这样做：

```
my ($dev, $ino, undef, undef, $uid, $gid) = stat($file);
```

最后一个列表元素可以是一个数组或散列：

```
($a, $b, @rest) = split;
my ($a, $b, %rest) = @arg_list;
```

实际上，可以在要赋值的列表中任意位置放入一个数组或散列，不过列表中的第一个数组或散列会“吸收”其余的所有值，它后面的部分将设置为未定义值。这在local或my中可能很有用，你可能希望在这些声明中将数组初始化为空。

甚至可以赋值为空列表：

```
() = funkshun();
```

这会在列表上下文中调用函数，不过将丢掉返回值。如果只是调用函数而没有赋值，则会在void上下文中调用这个函数，这是一种标量上下文，可能会使函数有完全不同的表现。

标量上下文中的列表赋值将返回赋值等式右边表达式所生成的元素个数：

```
$x = ( ($a, $b) = (7,7,7) ); # 将$x设置为3，而不是2
$x = ( ($a, $b) = funk() ); # 将$x设置为funk()返回的元素个数
$x = ( () = funk() );      # 也将$x设置为funk()返回的元素个数
```

如果想在布尔上下文中完成列表赋值，这会很方便，因为大多数列表函数完成时都会返回一个null列表，这在赋值时会生成一个0，而0将解释为false。可以如下在while语句中使用：

```
while (($login, $password) = getpwent) {
    if (crypt($login, $password) eq $password) {
        print "$login has an insecure password!\n";
    }
}
```

## 数组长度

要得到数组@days中的元素个数，可以在标量上下文中计算@days，如下所示：

```
@days + 0;      # 隐式强制@days到标量上下文
scalar(@days)   # 显式强制@days到标量上下文
```

注意，这只适用于数组。对于一般的列表值并不适用。前面已经提到，在标量上下文中计算逗号分隔的列表时，会返回最后一个值，这类似于C的逗号操作符。不过，由于在Perl中你几乎不需要知道列表的长度，所以这不算是问题。

\$#days与@days的标量计算紧密相关。它会返回数组最后一个元素的下标，即数组长度减1，因为元素下标从0计起。为\$#days赋值会改变数组的长度。采用这种方法来缩短一个数组时，将撤销缩短部分的值。如果一个数组可能很大，通过提前扩展数组可以提高效率（也可以为超出数组末尾的元素赋值，通过这种方式扩展数组）。如果将数组赋值为空列表()，这会把数组截断为不包含任何元素。下面两个语句是等价的：

```
@whatever = ();
$#whatever = -1;
```

下面的表达式总为true：

```
scalar(@whatever) == $#whatever + 1;
```

截断一个数组并不会回收其内存。必须调用undef(@whatever)（或者让它超出作用域），

才能将其内存放回进程内存池。可能无法将这个内存回收到系统内存池，因为几乎没有操作系统支持这一点。

## 散列

前面已经说过，散列是一种有趣的数组，要用键字符串而非数字来查找相应的值。散列定义了键与值之间的关联，所以散列通常也被勤奋的人（不拒绝多敲几个字母）称为关联数组（associative arrays）。

在Perl中实际上没有散列直接量的概念，不过如果为散列赋一个正常的列表，将取列表中的每一对值指示一个键/值关联：

```
my %map = ("red",0xff0000,"green",0x00ff00,"blue",0x0000ff);
```

这与以下代码有相同的效果：

```
my %map; # 未初始化的散列为空
$map{red} = 0xff0000;
$map{green} = 0x00ff00;
$map{blue} = 0x0000ff;
```

在键/值对之间使用=>操作符会更可读。=>操作符就是逗号的一个同义词，不过看起来更有特点，还可以引用它左边的裸标识符（如以上大括号中的标识符），这样可以很方便地完成很多操作，包括初始化散列变量：

```
my %map = (
    red => 0xff0000,
    green => 0x00ff00,
    blue => 0x0000ff,
);
```

或者初始化匿名散列引用作为记录：

```
my $rec = {
    NAME => "John Smith",
    RANK => "Captain",
    SERNO => "951413",
};
```

或者使用命名参数来调用复杂的函数：

```
my $field = radio_group(
    NAME => "animals",
    VALUES => ["camel", "llama", "ram", "wolf"],
    DEFAULT => "camel",
    LINEBREAK => "true",
    LABELS => \%animal_names,
);
```



不过这有点扯远了。再回到散列上来。

列表上下文中可以使用散列变量（%hash），在这里它会将其所有键/值对内插到列表中。不过，尽管散列采用一种特定的顺序初始化，但这并不表示值仍按这个顺序取出。散列在内部使用散列表实现，以加快查找的速度，这说明散列项存储的顺序取决于散列表中用来计算位置的内部散列函数，而不是依赖于其他有意思的特性。所以返回这些散列项时，会采用一种看起来有些随机的顺序（当然，各个键/值对中的两个元素顺序是正确的）。可以参见第27章的keys函数一节，其中给出了如何指定输出顺序的几个例子。

在标量上下文计算一个散列变量时，只有当这个散列中包含有键/值对时它才会返回一个true值。如果确实有键/值对，返回的值将是一个字符串，其中包含已用的桶（buckets）数和已分配的桶数，二者用一个斜线分隔。这个返回值用处很有限，如果要查看Perl的（已编译）散列算法在处理你的数据集时效率如何，此时可以用到这个返回值。例如，散列中有10 000项，不过在标量上下文计算%HASH时得到“1/8”，这说明只用到8个桶中的一个。这意味着这个桶包含了全部的10 000个散列项。我们可不希望这样。

要查看散列中的键数，可以在标量上下文中使用keys函数：

```
scalar(keys(%HASH))
```

通过在大括号中指定多个键（用逗号分隔），可以计算一个多维散列。列出的键将连接在一起，用\$;（\$SUBSCRIPT\_SEPARATOR）的内容分隔，其默认值为chr(28)。得到的结果串将用作散列的具体键。下面这两行的作用是一样的：

```
$people{ $state, $county } = $census_results;  
$people{ join $; => $state, $county } = $census_results;
```

这个特性原来是为了支持a2p（awk-Perl转换器）而实现的。如今，通常会使用一个真正的（嗯，更真实的）多维数组（参见第9章）。有一种情况可能还会用到这个特性，如果散列关联到不支持多维键的外部文件，如DBM文件，就要使用这个特性。

不要把模拟多维散列与片段混为一谈。一个表示标量值（模拟多维散列），另一个表示列表值（片段）：

```
$hash{ $x, $y, $z } # 单个值  
@hash{ $x, $y, $z } # 包含3个值的一个片段
```

## 类型团和文件句柄

Perl使用一个名为类型团（typeglob）的特殊类型来保存整个符号表记录。符号表记录\*foo包含\$foo、@foo、%foo、&foo，以及foo的多种其他表示所对应的值。类型团的类型前缀是一个\*，因为这表示所有类型。

类型团（或引用）的一种用法是传递或存储文件句柄，这在Perl提供文件句柄引用之前尤其常用。如果想保存一个裸字文件句柄，可以这样做：

```
$fh = *STDOUT;
```

或者也可以保存为一个真正的引用，如下：

```
$fh = \*STDOUT;
```

或者可以访问该符号表记录中的文件句柄部分：

```
$fh = *STDOUT{IO};
```

原先这是创建局部文件句柄的首选方法。例如：

```
sub newopen {  
    my $path = shift;  
    local *FH;      # 不是my() 也不是our()  
    open(FH, '<', $path) || return undef;  
    return *FH;     # 不是 \*FH!  
}  
$fh = newopen("/etc/passwd");
```

不过，如今更好的办法往往是让Perl选择一个文件句柄，并为你填入一个空变量：

```
sub newopen {  
    my $path = shift;  
    open(my $fh, '<', $path) || return undef;  
    return $fh;  
}  
$fh = newopen("/etc/passwd");
```

现在类型团的主要用法是为一个符号表记录指定别名（即另一个符号表记录）。可以把别名看作是一个“外号”。如果有：

```
*foo = *bar;
```

这会使名为“foo”的所有变量成为名为“bar”的相应变量的同义词。也可以赋一个引用来指定类型团中一个变量的别名：

```
*foo = \ $bar;
```

就会将\$foo作为\$bar的一个别名，但是不会使@foo成为@bar的别名，也不会使%foo作为%bar的别名。所有这些只影响全局（包）变量，不能通过符号表记录访问词法作用域变量。像这样为全局变量指定别名看起来有点傻，不过实际上整个模块导入/导出机制就是基于这个特性建立的，因为没有要求说只能为你的命名空间中的符号建立别名。下面这个语句：

```
local *Here::blue = \ $There::green;
```

会临时使`$Here::blue`成为`$There::green`的一个别名，不过它不会让`@Here::blue`成为`@There::green`的别名，也不会使`%Here::blue`成为`%There::green`的别名。幸运的是，所有这些复杂的类型团管理都是隐藏的，不需要你看到。参见第8章的“处理引用”和“符号表引用”，第10章中的“符号表”，以及第11章，将会更多地讨论类型团和导入。

## 输入操作符

这里会讨论一些输入操作符，因为它们会解析为项。有时我们把这些输入操作符称为伪直接量（*pseudoliterals*），因为它们在很多方面非常类似于加引号的字符串（解析为列表操作符的输出操作符（如`print`）将在第27章讨论）。

### 命令输入（反引号）操作符

首先来看命令输入操作符，这也称为反引号操作符，因为它的用法如下：

```
$info = `perldoc $module`;
```

用反引号（从技术上讲，也就是重音符）引起的字符串首先会像双引号字符串一样完成变量内插。再由系统将其结果解释为一个命令行，这个命令的输出就是这个伪直接量的值（这种行为与UNIX shells中的一个类似操作符相仿）。在标量上下文中，会返回由所有输出构成的一个字符串。在列表上下文中，将返回一个值列表，每个值对应一行输出（可以设置`$/`来使用一个不同的行终止符）。

每次执行伪直接量时都会执行这个命令。命令的数值状态值保存在`$?`中（`$?`的解释见第25章，这也称为`$CHILD_ERROR`）。与这个命令的`csh`版本不同，对返回数据不完成任何转换，换行符还是换行符。与所有其他shell中不同，Perl中的单引号不会隐藏命令中的变量名，即不会阻止解释这些变量名。要向shell传递一个`$`，需要用一个反斜线来隐藏。以上`finger`例子中的`$user`会由Perl内插，而不是由shell处理（因为命令会经过shell处理，有关的安全问题参见第20章）。

反引号的一般形式为`qx//[表示“quoted execution”（引号执行）]`，这个操作符的做法与正常的反引号完全相同。只需要选择你的引号字符。与引号伪函数类似，如果刚好选择了单引号作为定界符，这个命令字符串不会完成双引号内插：

```
$perl_info = qx(ps $$); # 这是Perl的$$  
$shell_info = qx'ps $'$; # 这是shell的$$
```

### 行输入（尖角）操作符

最常使用的输入操作符就是行输入操作符，这也称为尖角操作符或`readline`函数（因为这正是它在内部调用的函数）。计算尖括号中的一个文件句柄（例如`STDIN`）时，会从关



联的文件句柄得到下一行（这也包括换行符，所以根据Perl的真值规则，读行操作总会返回true，直到文件末尾，在文件末尾会返回一个未定义的值，这会得到false）。正常情况下，你要把输入值赋给一个变量，不过有一种情况会发生自动赋值。当且仅当一个while循环的条件语句中只有一个行输入操作符时，这个值会自动赋给特殊变量\$<sub>0</sub>。再测试所赋的这个值，查看它是否已定义（在你看来这可能有些奇怪，不过你会经常使用这个构造，所以很有必要好好学学）。总之，以下代码行是等价的：

```
while (defined($0 = <STDIN>)) { print $0 }      # 最长的方式
while ($0 = <STDIN>) { print }                     # 显式赋值给$0
while (<STDIN>) { print }                           # 简短方式
for (;<STDIN>;) { print }                           # 伪装的while循环
print $0 while defined($0 = <STDIN>);             # 长语句修饰符
print while $0 = <STDIN>;                           # 显式赋值给$0
print while <STDIN>;                                # 短语句修饰符
```

要记住，这个特殊魔法需要用到一个while循环。如果在别处使用输入操作符，要想保留这个值，必须显式地用结果赋值：

```
while (<FH1> && <FH2>) { ... }                      # 错误：两个输入都会丢掉
if (<STDIN>) { print }                               # 错误：会打印$0原来的值
if ($0 = <STDIN>) { print }                           # 不太好：不会测试是否已定义
if (defined($0 = <STDIN>)) { print }                 # 最好的方法
```

在一个\$<sub>0</sub>循环中隐式地对\$<sub>0</sub>赋值时，这是一个名为\$<sub>0</sub>的全局变量，而不局限于while 循环。可以采用以下方式保护\$<sub>0</sub>的现有值：

```
while (local $0 = <STDIN>) { print } # 全局$0的临时值
```

或者采用以下方式：

```
while (my $0 = <STDIN>) { print } # 一个新词法作用域变量$0
```

循环结束时，之前的值会恢复。除非用my或state声明，否则\$<sub>0</sub>仍是一个全局变量，所以从这个循环内部调用的函数仍能访问它（有意或无意）。也可以避免这样做，把它声明为词法作用域变量。更好的办法是，为你的词法作用域变量指定一个合适的名字：

```
while (my $line = <STDIN>) { print $line } # 现在是私有的
```

这两个while循环仍然会隐式地测试赋值结果是否已定义，因为my、state、local和our不会改变解析器所看到的赋值方式。文件句柄STDIN、STDOUT和STDERR都是预定义而且预打开的。可以用open或sysopen函数创建其他文件句柄。有关的详细信息请参见第27章中这些函数的文档。

在上面的while循环中，我们是在标量上下文中计算这个行输入操作符，所以操作符会单独返回各行。不过，如果在列表上下文中使用这个操作符，会返回一个列表，其中包含其余的所有行，每个列表元素对应一行。这样一来很容易建立一个很大的数据空间，所以使用这个特性时要当心：

```
$one_line = <MYFILE>; # 得到第一行
@all_lines = <MYFILE>; # 得到其余各行
```

列表形式的输入操作符并没有while魔法，因为while循环的条件总是提供标量上下文（与所有条件语句一样）。

在尖角操作符中使用空文件句柄很特殊；它会模拟典型UNIX过滤器程序（如sed和awk）的命令行行为。从<>读行时，会魔法般地提供命令行中提到的所有文件中的所有行。如果没有提到任何文件，它会提供标准输入，所以你的程序很容易插入到一个进程管道的中间。

做法如下：第一次计算<>时，会检查@ARGV数组。如果为空（null），\$ARGV[0]设置为“-”，打开时就会提供标准输入。然后将@ARGV数组作为一个文件名列表进行处理。更确切地讲，以下循环：

```
while (<>) {
    ...                # 每一行的代码
}
```

等价于以下类Perl伪代码：

```
@ARGV = ("-") unless @ARGV; # 如果为空则假设为STDIN
while (@ARGV) {
    $ARGV = shift @ARGV;      # 每次会缩短@ARGV
    if (!open(ARGV, '<', $ARGV)) {
        warn "Can't open $ARGV: $!\n";
        next;
    }
    while (<ARGV>) {
        ...                # 每一行的代码
    }
}
```

不过前一种写法没有后面的伪代码那么啰唆，而且确实能正常工作。它会逐一处理数组@ARGV中的元素，将当前文件名放入全局变量\$ARGV。这还在内部使用了特殊文件句柄ARGV——<>就是写法更明确的<ARGV>的同义词，这是一个有魔法的文件句柄（上面的伪代码不能运行，因为它会把<ARGV>当作没有魔法的文件句柄）。

可以修改第一个<>之前的@ARGV，只要数组最后包含有你确实想要的文件名列表。由于Perl在这里使用了常规的open函数，只要遇到文件名“-”，就会处理为标准输入，你将自动得到open的其他更神秘的特性（如忽略文件名中的前导或末尾空白符，或者打开一个名为“gzip -dc < file.gz |”的“文件”）。行号(\$.)会继续（而不是每个文件单计行号），就好像输入是一个大文件一样（不过，对于如何重置每个文件的行号，请参见第27章中关于eof的例子）。

如果想把@ARGV设置为你自己的文件列表，这是完全可以的：

```
# 如果没有给定参数，默认为README文件
@ARGV = ("README") unless @ARGV;
```

如果想为脚本传入开关，可以使用某个`Getopt::*`模块，或者在前面加一个循环，如下所示：

```
while (@ARGV and $ARGV[0] =~ /^-/) {
    $_ = shift;
    last if /^--$/;
    if (/^-D(.*)/) { $debug = $1 }
    if (/^-v/) { $verbose++ }
    ...                # 其他开关
}
while (<>) {
    ...                # 每一行的代码
}
```

`<>`符号只会返回一次`false`。如果在此之后再次调用，它会认为你在处理另一个`@ARGV`列表，如果没有设置`@ARGV`，它将从`STDIN`读取输入。

如果尖角操作符中的字符串是一个标量变量（例如，`<$foo>`），这个变量包含一个间接文件句柄（indirect filehandle），可能是将从中得到输入的文件句柄名，或者是这样一个文件句柄的引用。例如：

```
$fh = \*STDIN;
$line = <$fh>;
```

或：

```
open(my $fh, '<', "data.txt");
$line = <$fh>;
```

## 文件名聚团操作符

你可能想知道，如果在尖角操作符中放一些更有意思的东西，行输入操作符会怎么做。它会变成一个不同的操作符。如果尖括号中的字符串不是一个文件句柄名或标量变量（甚至只是多加了空格），它就会被解释为一个要“聚团”的文件名模式<sup>注20</sup>。会对当前目录（或作为文件团模式一部分指定的目录）中的文件完成模式匹配，将由这个操作符返回匹配的文件名。与行输入操作符一样，在标量上下文中会一次返回一个文件名，而在列表上下文中会一次全部返回。后一种用法更为常见；你会经常看到这样的代码：

```
@files = <*.xml>;
```

---

注20：文件团与前面提到的类型团没有任何关系，只不过它们都使用了表示通配符的`*`字符。采用这种方式使用时，`*`字符有一个外号叫做“团”。对于类型团，就是把符号表中同名的所有符号“聚团”。对于文件团，则是对一个目录中的文件名完成通配匹配，很多shell都会这么做。



与其他类型的伪直接量一样，首先会完成一级变量内插，不过不能写为<\$foo>，因为这是一个间接文件句柄（前面解释过）。在老版本的Perl中，程序员会插入大括号，强制作为一个文件团内插：<\${foo}>。如今，直接调用内部函数会更清晰，如glob(\$foo)（原本就应该这样）。所以如果不愿意为此重载尖角操作符，应该写为：

```
@files = glob("*.xml");
```

这是允许的。

不论使用glob函数还是用老式的尖括号，文件团操作符都会像行输入操作符一样提供while魔法，将结果赋给\$\_（这也是最初重载尖角操作符的根本原因）。例如，如果想改变所有C代码文件的权限，可以写为：

```
while (glob "*.c") {  
    chmod 0644, $_;  
}
```

这等价于：

```
while (<*.c>) {  
    chmod 0644, $_;  
}
```

在老版本的Perl中（以及更老的UNIX），glob函数最初实现为一个shell命令，这说明执行的开销相当大，更糟糕的是，不同系统上glob函数的工作并不完全一样。如今，这已经是一个内置函数，所以更可靠，另外执行速度也快得多。

当然，要完成上面的chmod命令，最简短（无疑最可读）的方法是使用文件团作为一个列表操作符：

```
chmod 0644, <*.c>;
```

开始一个新列表时，文件团才会计算其（内嵌）操作数。所有值都必须在操作符开始之前读入。在列表上下文中，这并不重要，因为你会自动得到所有值。不过，在标量上下文中，每次调用时，这个操作符会返回下一个值，或者如果所有值都读完，则返回一个false值。同样地，false只返回一次。所以，如果希望从一个文件团得到单个值，最好这样写：

```
($file) = <blurch*>;# 列表上下文
```

而不是：

```
$file = <blurch*>; # 标量上下文
```

因为前者会返回所有匹配的文件名，并重置操作符，而后者有可能返回文件名，也可能返回false。

如果想完成变量内插，最好使用`glob`操作符，因为原来的记法会导致与间接文件句柄记法混淆。从这里可以看到项和操作符之间的界限有点模糊：

```
@files = <$dir/*.ch>;           # 可行，但要避免这么做
@files = glob("$dir/*.ch");      # 调用glob作为函数
@files = glob $some_pattern;     # 调用glob作为操作符
```

最后一个例子中去掉了小括号，这是为了说明`glob`既可以用作函数（项），也可以用作一个一元操作符；也就是只取单个参数的一个前缀操作符。`glob`操作符是一个命名一元操作符，这只是我们下一章将要讨论的操作符中的一种。接下来，我们会介绍模式匹配操作符，同样地，模式匹配操作符解析时与项类似，而行为却像操作符。

# 一元和二元操作符

第2章我们讨论了表达式中可能用到各种类型的项，不过坦率地讲，单独来看这些项确实有些枯燥。就像那些热衷于聚会的家伙一样，很多项喜欢相互拉关系。一般的新项总是极力得到其他项的认同，并希望以各种不同的方式影响其他项，不过有很多种不同的社会交往方式，另外投入程度也有很多不同层次。在Perl中，这些关系要用操作符来表达。

学好社会学肯定有好处。

从数学的角度来看，操作符就是有特殊语法的普通函数。从语言学的角度看，操作符就是不规则动词。不过，语言学家会告诉你，语言中的不规则动词往往是最常用到的动词。从信息理论角度来讲这很重要，因为不规则动词往往更短，在产生和识别方面都更为高效。

从实用的角度来说，操作符很方便。

操作符有各种不同类型，这取决于它们的元数（有多少个操作数）、优先级（与周围的操作符相比，它们竞争操作数的能力）以及结合性（与相同优先级的操作符结合时，是从左向右处理还是从右向左处理）。

Perl操作符有3种元数：一元（unary）、二元（binary）和三元（trinary或ternary）。一元操作符往往是前缀操作符（除后自增和后自减操作符以外）<sup>注1</sup>。其他操作符都是中缀操作符，但列表操作符例外，它可以放在任意多个参数前面。不过大多数人认为列表操作符是常规函数，只是可以不加小括号。下面给出几个例子：

<code>! \$x</code>	# 一个一元操作符
<code>\$x * \$y</code>	# 一个二元操作符
<code>\$x ? \$y : \$z</code>	# 一个三元操作符
<code>print \$x, \$y, \$z</code>	# 一个列表操作符

注1： 不过你可能认为各种引号和括号是对项定界的强调操作符。



操作符的优先级控制了它绑定的紧密程度。有较高优先级的操作符会比优先级较低的操作符更早获取参数。小学数学里就有最直接的例子，乘法总是优先于加法：

2 + 3 \* 4 # 得到14，而不是20

有相同优先级的两个操作符的执行顺序由其结合性决定。在某种程度上，这些规则也同样遵循数学惯例：

2 \* 3 \* 4  
2 \*\* 3 \*\* 4  
2 != 3 != 4

# 表示 (2 \* 3) \* 4，左结合  
# 表示 2 \*\* (3 \*\* 4)，右结合  
# 不合法，无结合性

表3-1按优先级从高到低列出了Perl操作符的结合性和元数。

表3-1：操作符优先级

结合性	元数	优先类
无	0	项和列表操作符（左边）
左结合	2	->
无	1	++ --
右结合	2	**
右结合	1	! ~ \以及一元操作符+和-
左结合	2	= ~ ! ~
左结合	2	* / % x
左结合	2	+ - .
左结合	2	<< >>
右结合	0,1	命名一元操作符
无	2	< > <= >= lt gt le ge
无	2	== != <=> eq ne cmp ~~
左结合	2	&
左结合	2	^
左结合	2	&&
左结合	2	//
无	2	.. ...
右结合	3	?:
右结合	2	= += -= *= 等
左结合	2	, =>
右结合	0+	列表操作符（右边）
右结合	1	not
左结合	2	and
左结合	2	or xor

看起来要记的不同优先级太多了。嗯，你说的没错，确实有很多不同的优先级。幸运的是，对你来说有两个好消息。首先，这里定义的优先级通常与你的直觉是一致的，当然前提是思路不变态。其次，如果你有些神经质，可以加上额外的括号来缓解你的紧张情绪。

还有一个提示可能有帮助，从C借用的所有操作符仍保留相同的优先级关系（尽管C的优先级有些怪异）。这使得使用C和C++的人（甚至包括使用Java的人）学习Perl会更为容易。

下面各节将按优先级顺序介绍这些操作符。除了少数几个例外，所有这些操作符只处理标量值，而不处理列表值。如果有例外情况，即处理的是列表值，我们将特别指出。

尽管引用都是标量值，但这里大多数操作符用于引用时并没有太大意义，因为引用的数值只对Perl内部有意义。不过，如果一个引用指向某个类的一个对象，而且这个类允许重载，就能对这些对象调用这些操作符，如果这个类已经为该操作符定义了一个重载实现，它就会定义该操作符如何处理这个对象。例如，Perl中的复数就是这样实现的。关于重载的更多内容请参见第13章。

## 项和列表操作符（左边）

项（term）在Perl中优先级最高。项包括变量，引号和类引号操作符，小括号、中括号或大括号中的大多数表达式，以及将参数用括号括起的函数。实际上，从这个意义上讲，这并不是真正的函数，而只是因为参数两边加上了括号而相当于函数的列表操作符和一元操作符。不过，第27章的名字就是函数。

现在请特别注意。下面给出两个非常重要的规则，它们能大大简化问题，不过如果不当心有时可能会生成与直觉不符的结果。如果列表操作符（如`print`）或命名一元操作符（如`chdir`）后面的第一个token是一个左括号（忽略空白符），这个操作符及用括号括起的参数将有最高的优先级，就好像这是一个正常的函数调用。有这样一个规则：如果看起来像是一个函数调用，这就是一个函数调用。通过在括号前面加一个一元加号，可以让它看起来像一个非函数，从语义上讲其实这个加号什么都不做，甚至不会将参数转换为数值。

例如，由于`||`的优先级低于`chdir`，所以可以有：

<code>chdir \$foo</code>	<code>   die;</code>	<code># (chdir \$foo)    die</code>
<code>chdir(\$foo)</code>	<code>   die;</code>	<code># (chdir \$foo)    die</code>
<code>chdir (\$foo)</code>	<code>   die;</code>	<code># (chdir \$foo)    die</code>
<code>chdir +(\$foo)</code>	<code>   die;</code>	<code># (chdir \$foo)    die</code>

不过，由于`*`的优先级比`chdir`高，所以有：

<code>chdir \$foo * 20;</code>	<code># chdir (\$foo * 20)</code>
--------------------------------	-----------------------------------

```
chdir($foo) * 20;      # (chdir $foo) * 20
chdir ($foo) * 20;     # (chdir $foo) * 20
chdir +($foo) * 20;    # chdir ($foo * 20)
```

作为命名一元操作符的所有数值操作符也与此类似，如rand：

```
rand 10 * 20;          # rand (10 * 20)
rand(10) * 20;         # (rand 10) * 20
rand (10) * 20;        # (rand 10) * 20
rand +(10) * 20;       # rand (10 * 20)
```

如果没有小括号，列表操作符（如print、sort或chmod）的优先级可能很高，也可能很低，这取决于这个操作符出现在左边还是右边（这正是这一节标题中有一个“左边”的原因）。例如，在以下代码中：

```
my @ary = (1, 3, sort 4, 2);
print @ary;      # 打印1324
```

sort右边的逗号会先于sort计算，而左边的逗号会在sort之后计算。换句话说，列表操作符会把它后面的所有参数“团”在一起，这样一来，对于前面的表达式，它就相当于单独的一项。还要特别当心小括号：

```
# 这些语句会在完成print之前执行exit:
print($foo, exit);  # 这显然不是你想要的
print $foo, exit;   # 也不是这样

# 这些语句会在执行exit之前完成print:
(print $foo), exit; # 这是你想要的
print($foo), exit;  # 或者这样
```

有一种情况最容易出问题，你可能使用小括号对数学参数分组，不过忘记了括号也用于对函数参数分组：

```
print ($foo & 255) + 1, "\n"; # 打印($foo & 255)
```

看上去它的做法与你期望的可能不一样<sup>注2</sup>。幸运的是，如果启用了警告，这种错误一般会生成警告，如“Useless use of addition (+) in void context”（在void上下文使用了无用的加号(+)）和“print (...) interpreted as function”（print (...)解释为函数）。第二个警告会提醒你：这两个括号已经界定了参数表，而它后面的部分(+ 1, “\n”)不是参数。应该把它写为：

```
print(($foo & 255) + 1, "\n"); # 打印($foo & 255)+1
```

do {}和eval {}构造也解析为项，另外子例程和方法调用、匿名数组和散列构造器[]和{}，以及匿名子例程构造器sub {}都会解析为项。

---

注2： 正是因为这一点，我们在Perl 6中修正为与你期望的相一致。可惜的是，我们不能在Perl 5完成这个修正，否则会破坏大量已有代码。



## 箭头操作符

与C和C++中一样，一元->操作符是一个用于解引用的中缀操作符。如果右边是一个[...]数组下标、{...}散列下标或(...)子例程参数表，左边就必须分别是一个数组、散列或子例程的引用<sup>注3</sup>：

```
$aref->[42]           # 数组解引用
$href->{"corned beef"} # 散列解引用
$sref->(1,2,3)         # 子例程解引用
```

在一个左值（可赋值）上下文中，如果左边不是一个引用，它就必须是一个能够存放硬引用的位置，在这种情况下，会为你自动生成（autovivified）这样一个引用。

```
$aref->[42] = 'Huh!';      # 自动生成$aref中的数组
$href->{"corned beef"} = 0; # 自动生成$href中的散列
```

不论哪一种情况，都会用所赋的值创建这个新的数组或散列元素。有关的更多内容（以及意外自动生成的有关警告）请参见第8章。

如果箭头右边不是这样一些括号，这就是某种方法调用。右边必然是一个方法名（或者是一个简单的标量变量，其中包含方法名或一个方法引用），左边必须计算为一个对象（一个被“祝福”的引用）或者一个类名（也就是一个包名）（译者注：将类名与引用相结合就称为“祝福”（bless）一个对象）：

```
my $yogi = Bear->new( "Yogi" );      # 类方法调用
$yogi->swipe('picnic basket');        # 对象方法调用
```

方法名可以用一个包名来限定，指示在哪个类中开始搜索这个方法，或者用特殊包名SUPER::来指示应当从父类开始搜索。详细内容见第12章。

## 自增和自减

++和--操作符的作用与C中相同。也就是说，放在一个变量前面时，它们会在返回值之前使变量增1或减1；如果放在变量后面，则会在返回值之后再使变量增1或减1。例如，\$a++会让标量变量\$a的值增1，不过将在其递增之前返回值。类似的，--\$b{(/(\w+)/)[0]}将散列%b中由默认搜索变量(\$\_)中第一个“单词”索引的元素减1，并在递减之后返回值<sup>注4</sup>。需

注3： 这可以是一个符号引用，不过这只有在未启用strict约束时才有效。否则，这将是一个硬引用。

注4： 嗯，这也不完全正确。我们这么说只是想引起你的注意。这个表达式的工作是这样的：首先，模式匹配使用正则表达式\w+在\$\_中查找第一个单词。两边的括号使得这个单词作为一个单元元素列表值返回，因为模式匹配在列表上下文中完成。这个列表上下文由列表片段操作符(...)[0]提供，它会返回列表中的第一个（也是唯一一个）元素。这个值用作散列的键，然后对应该键存储的散列值自减，并返回。一般来讲，遇到一个复杂的表达式时，要从最内层开始向外分析，表达式就是按这个顺序计算的。

要说明，与C中一样，Perl没有定义变量何时递增或递减。你只知道它会在返回值之前或之后某个时间完成。这也说明在同一个语句中将一个变量修改两次会导致不确定的行为。要避免类似这样的语句：

```
$i = $i++;  
print ++$i + $i++;
```

Perl不能保证这种代码的结果。

自增操作符还有一点额外的魔法。如果让一个数值变量自增，或者这个变量在一个数值上下文中使用，就会完成正常的自增。不过，如果变量自设置以来只在字符串上下文中使用，而且它的值是可以匹配模式/<sup>^</sup>[a-zA-Z]\*[0-9]\*\z/的非空串，那么就会作为一个字符串来完成自增，保持各字符仍在其范围内（含上下界）：

```
my $foo;  
$foo = "99"; print ++$foo; # 打印"100"  
$foo = "a9"; print ++$foo; # 打印"b0"  
$foo = "Az"; print ++$foo; # 打印"Ba"  
$foo = "zz"; print ++$foo; # 打印"aaa"
```

未定义的值都会处理为数值，具体地会在自增前转换为0，所以一个未定义值（undef）的后自增会返回0而不是undef。

写这本书时，这种有魔法的自增还没有扩展到Unicode字母和数字，不过将来有可能做到。

不过，自减操作符并没有这种魔法。

## 指数

二元\*\*是指数操作符。需要注意，它甚至比一元负号绑定更为紧密，所以-2\*\*4是-(2\*\*4)，而不是(-2)\*\*4。这个操作符使用C的pow(3)函数实现，它在内部会处理浮点数。指数操作符使用对数计算，这说明它处理的是小数幂，不过有时得到的结果可能与直接相乘生成的结果不完全相同。

## 表意一元操作符

大多数一元操作符都有名字（见本章后面的“命名一元操作符和文件测试操作符”），不过有些操作符非常重要，因此有自己的特殊符号表示。所有这些操作符都与“取非”（否定）有关。这要怪数学家。

一元!操作符完成逻辑非操作；也就是“否（not）”。可以参见not（这是一个较低优先级的逻辑非操作符）。对一个操作数取非时，若该操作数为false（数值0、字符串"0"、空串或未定义），取非得到的值则为true。如果操作数为true，取非结果则为false(“”）。



一元一操作符完成算术取非操作（如果操作数为数值）。如果操作数是一个标识符，则会返回一个字符串，由一个负号连接这个标识符构成。否则，如果字符串以一个加号或减号开头，就会返回以相反符号开头的一个字符串。根据这些规则可以得到：`-bareword`等价于`"-bareword"`<sup>注5</sup>。不过，如果字符串以一个非字母字符开头（除“+”或“-”以外），Perl就会尝试将这个字符串转换为数值，并完成数值取非操作。如果这个字符串无法转换为一个数值，Perl会给出警告“Argument “the string” isn't numeric in negation (-)”，指示在取非操作中参数不是一个数值。

一元~操作符完成按位取非操作；也就是说，求1的补码。例如，`0666 & ~027`会得到0640。根据定义，这种做法是不可移植的（受机器的字长限制）。例如，在一个32位机器上，`~123`是4294967172，而在一个64位机器上，这是18446744073709551492。不过这一点你已经知道。

你可能不知道的是，如果~的参数恰好是一个字符串而不是一个数字，会返回一个等长的字符串，不过字符串的所有位都变为其补码。这是一种很快捷的方法，可以采用一种可移植的方式一次翻转很多位，因为它不依赖于计算机的字长。后面我们还会介绍位逻辑操作符，它们同样提供了面向字符串的“版本”。

对字符串求补码时，如果所有字符的序数值都小于256，其补码也小于256。不过，如果不是这样，所有字符都将使用32位或64位补码，这取决于你的机器的体系结构。例如，在32位机器上表达式`~"\x{3B1}"`是`"\x{FFFF_FC4E}"`，在64位机器上则是`"\x{FFFF_FFFF_FFFF_FC4E}"`。

一元+操作符没有任何语义作用，甚至对字符串也没有影响。它只是在语法上用来将函数名与加括号的表达式分隔开（否则可能会把它错误地解释为一个完整的函数参数表）可以参见“项和列表操作符（左边）”一节给出的例子。如果从另一个角度来考虑，可以认为+否定了括号将前缀操作符转变为函数的作用。

一元\操作符会它后面的内容创建一个引用。用于列表时，它会创建一个引用列表。详细内容请参见第8章中“反斜线操作符”一节。不要把这种行为与字符串中反斜线的行为弄混了，尽管这两种形式都有某种否定的含义，会阻止后面内容的正常解释。这种相似性并非偶然。

## 绑定操作符

二元操作符`=~`会把一个字符串表达式绑定到一个模式匹配、替换或转译（也称为转换）操作。否则如果没有指定这样一个字符串表达式，这些操作（模式匹配、替换或转换）就会搜索或修改`$_`（默认变量）中包含的字符串。要绑定的字符串放在左边，操作符本身放在

---

注5： 这对于Tk程序员最为有用，因为他们最先采纳这个约定。



右边。在标量上下文中，返回值通常指示右边的操作符是否成功，因为绑定操作符本身并不做什么。不过有一个例外，对替换（`s///`）或转换（`y///`，`tr///`）使用`/r`修饰符时，这会返回修改后字符串的一个副本。列表上下文中的行为则取决于具体的操作符。

如果右边的参数是一个表达式而不是一个模式匹配、替换或转换，这将在运行时解释为一个模式匹配。也就是说，`$_ =~ $pat`等价于`$_ =~ /$pat/`。相对于显式搜索来说，这样效率要低一些，因为必须检查模式，而且每次计算表达式时可能还需要重新编译。可以使用`qr/`（引用表达式）操作符预编译原模式，这样就能避免重新编译。

二元操作符`!~`与`=~`很类似，只不过是返回值会逻辑取非。如果二元操作符“`!~`”对一个非破坏性替换或转换使用`/r`修饰符，这是一个语法错误。除此以外，以下表达式在功能上是等价的：

```
$string !~ /pattern/  
!( $string =~ /pattern/ )  
not $string =~ /pattern/
```

我们说过返回值会指示是否成功，不过有很多不同类型的成功。除非使用`/r`修饰符要求返回其结果，否则替换只会返回成功匹配数，转换也是如此（实际上，转换操作符通常就用来统计字符数）。由于非0结果都认为是`true`，所以只要结果非0，操作就成功。最特殊的真值是模式的列表赋值：在列表上下文中，模式匹配会返回与模式中括号匹配的子串。不过，同样地，根据列表赋值规则，如果有匹配而且已赋值，列表赋值本身就会返回`true`，否则返回`false`。所以有时你可能会看到类似这样的代码：

```
if (my ($k,$v) = $string =~ m/(\w+)= (\w*)/) {  
    print "KEY $k VALUE $v\n";  
}
```

下面来分解这段代码。`=~`的优先级比`=`高，所以`=~`先处理。`=~`将`$string`绑定到右边的模式匹配，这会在字符串中扫描形如`KEY=VALUE`的所有出现。这是在列表上下文中，因为它在列表赋值的右边。如果能够完成模式匹配，会返回一个列表，赋至`$k`和`$v`，这是`my`创建的新变量。列表赋值本身在标量上下文中，所以它会返回2，也就是赋值等式右边的值个数。2会解释为`true`，因为标量上下文也是一个布尔上下文。如果匹配失败，没有赋任何值，则会返回0，也就是`false`。

关于匹配策略的更多内容请参见第5章。

## 乘除操作符

Perl提供了一些类C的操作符`*`（乘法）、`/`（除法）和`%`（求余或取模）。`*`和`/`的工作与你预想的一样，就是将两个操作数相乘或相除。除法按浮点数处理，除非你使用了`integer`、`bigint`、`bigrat`或`bignum pragma`模块。`%`操作符在找到整数除法的余数之前会将

其操作数转换为整数（不过，如果有必要，它会按浮点数完成这个整数除法，所以在大多数32位机器上操作数最多15位）。假设两个操作数名为\$a和\$b。如果\$b是正数，\$a % \$b的结果就是\$a减去\$b不大于\$a的最大倍数（这说明，结果总在0 .. \$b-1范围内）。如果\$b是负数，\$a % \$b的结果就是\$a减去\$b不小于\$a的最小倍数（这说明结果总在\$b+1 .. 0范围内）。

如果声明了use integer, %允许你直接访问C编译器实现的取余操作符。这个操作符对负操作数的处理没有明确定义，不过它执行的速度更快。

二元操作符x是重复操作符。实际上，这是两个操作符。在标量上下文中，它会返回一个字符串，由左操作数重复多次连接而成（次数由右操作数指定）。为了做到向后兼容，如果左参数不在括号中，也会在列表上下文中完成处理。

```
print "-" x 80;           # 打印一行短横线
print "\t" x ($tab/8), " " x ($tab%8); # 制表符
```

在列表上下文中，如果左操作数是一个用括号括起的列表，或者是qw/STRING/构成的列表，x将作为一个列表重复操作符而不是字符串重复操作符。如果要将一个长度不确定的数组中所有元素初始化为相同的值，这会很有用：

```
my @ones = (1) x 80; # 80个1构成的列表
@ones = (5) x @ones; # 设置所有元素都为5
```

类似地，还可以使用x初始化数组和列表片段：

```
my %hash;
my @keys = qw(perls before swine);
@hash{@keys} = ("") x @keys;
```

需要指出，@keys既在赋值等式左边用作为一个键列表，同时也在赋值等式右边用作为一个标量值（返回数组长度）。前面的例子对%hash有同样的效果：

```
$hash{perls} = "";
$hash{before} = "";
$hash{swine} = "";
```

## 加减操作符

很奇怪，Perl也有常规的+（加法）和-（减法）操作符。如果有必要，这两个操作符都会将其参数由字符串转换为数值，并返回一个数值结果。

另外，Perl还提供了.操作符，这个操作符会完成字符串连接。例如：

```
my $almost = "Fred" . "Flintstone"; # 返回FredFlintstone
```

需要说明，Perl在连接的字符串之间没有加空格。如果你想加空格，或者需要连接多个字

字符串，可以使用join操作符（见第27章）。不过，大多数情况下，人们都会在双引号字符串中隐含地完成连接：

```

my $fullname = "$firstname $lastname";

```

# 移位操作符

移位操作符（<<和>>）将左参数左移（<<）或右移（>>）一定位数（由右参数指定），再将其返回。两个参数都应当是整数。例如：

```

1 << 4;      # 返回16
32 >> 4;     # 返回2

```

不过要当心。对于大数（或负数），取决于你的机器会用多少位表示整数，移位的结果可能有所不同。可以用bigint pragma避免这个限制。

```

use v5.14;
say 500 << 20;      # 打印524288000
say 500 << 200;     # （只）打印128000

use bigint;
say 500 << 200;
803469022129495137770981046170581301261101496891396417650688000

```

# 命名一元操作符和文件测试操作符

第27章介绍的一些“函数”实际上是一元操作符。表3-2列出了所有命名一元操作符。

表3-2：命名一元操作符

-X（文件测试）	fileno	lock	setnetent
abs	getc	log	setprotoent
alarm	getgrgid	lstat	setservent
caller	getgrnam	my	shift
chdir	gethostbyname	oct	sin
chomp	getnetbyname	ord	sleep
chop	getpeername	our	sqrt
chr	getpgrp	pop	srand
chroot	getprotobyname	pos	stat
close	getpwnam	prototype	state
closedir	getpwuid	quotemeta	study
cos	getsockname	rand	tell
dbmclose	glob	readdir	telldir
defined	gmtime	readline	tied



delete	hex	readlink	uc
do	int	readpipe	ucfirst
each	keys	ref	umask
eof	lc	reset	undef
eval	lcfirst	rewinddir	untie
exists	length	rmdir	values
exit	local	scalar	write
exp	localtime	sethostent	any (\$) sub
fc			

与列表操作符不同，一元操作符比一些二元操作符优先级更高。例如：

```
sleep 4 | 3;
```

不会休眠7s。它会休眠4s，然后得到sleep的返回值（通常为0），再与3完成位或操作，就好像这个表达式加了括号一样：

```
(sleep 4) | 3;
```

可以与下面的代码比较：

```
print 4 | 3;
```

这确实会先取4和3的位或结果值（也就是7），然后再打印，就像是写为：

```
print (4 | 3);
```

这是因为，print是一个列表操作符，而不是简单的一元操作符。一旦了解哪些操作符是列表操作符，就能将一元操作符和列表操作符轻松地区分开。如果有疑问，完全可以使用括号把命名一元操作符转变为一个函数。要记住，如果看起来像是一个函数，那么这就是一个函数。

关于命名一元操作符还有一点很有意思，如果没有提供参数，很多一元操作符会默认处理\$<sub>0</sub>。不过，如果忽略了参数，但是命名一元操作符后面的token看起来像是一个参数的开头，就会把Perl搞糊涂，因为它想要的是一个项。只要Perl的词法分析器遇到表3-3所列的一个字符，词法分析器会根据希望得到项还是操作符来返回不同的token类型。

表3-3：二义性字符

字符	操作符	项
+	加法	一元正号
-	减法	一元负号
*	乘法	*typeglob
/	除法	/pattern/

表3-3：二义性字符（续）

字符	操作符	项
<	小于，左移	<HANDLE>, <<END
.	连接	0.3333
?	?:	?pattern? （过时）
%	取余	%hash
&	&, &&	&subroutine

所以一个典型的错误是：

```
next if length < 80;
```

在这里，对于解析器来说，<看起来就像是<>输入符号（一个项）的开头，而不是你希望的“小于”（操作符）。这一点没有办法修正。如果你实在太懒，连输入两个字符（\$  ）都不愿意，可以使用以下做法：

```
next if length() < 80;
next if (length) < 80;
next if 80 > length;
next unless length >= 80;
```

希望得到一个项时，如果有一个负号后面跟一个字母，则总会解释为一个文件测试（file test）操作符。文件测试操作符是一个一元操作符，只有一个参数，这可以是一个文件名或是一个文件句柄，它会测试相关的文件，来看有关这个文件的某个判断是否为真。如果省略参数，它会测试\$  ，不过-t除外，-t会测试STDIN。除非有另外的文档说明，否则会返回1表示true，返回""表示false，或者如果文件不存在或不可访问，则返回未定义值。当前已经实现的文件测试操作符如表3-4所列。

表3-4：文件测试操作符

操作符	含义
-r	文件由有效的UID/GID可读
-w	文件由有效的UID/GID可写
-x	文件由有效的UID/GID可执行
-o	文件由有效的UID所有
-R	文件由真实的UID/GID可读
-W	文件由真实的UID/GID可写
-X	文件由真实的UID/GID可执行
-O	文件由真实的UID所有
-e	文件存在
-z	文件大小为0
-s	文件大小不为0（返回大小）

表3-4：文件测试操作符（续）

操作符	含义
-f	文件是一个普通文件
-d	文件是一个目录
-l	文件是一个符号链接
-p	文件是一个命名管道（FIFO）
-S	文件是一个套接字
-b	文件是一个块特殊文件
-c	文件是一个字符特殊文件
-t	为tty打开文件句柄
-u	文件setuid位已设置
-g	文件setgid位已设置
-k	文件sticky位已设置
-T	文件是一个文本文件
-B	文件是一个二进制文件（与-T对应）
-M	（启动时）文件自修改以来的年龄（天数）
-A	（启动时）文件自最后一次访问以来的年龄（天数）
-C	（启动时）文件自索引节点修改以来的年龄（天数）

这些操作符不遵循前面介绍的“看起来像函数规则”。也就是说，操作符后面的开始括号不会影响后面哪些代码将构成参数。例如，这意味着，`-f($file).".bak"`等价于`-f"$file.bak"`。把开始括号放在操作符前面可以将它与后面的代码分开（当然，这只适用于优先级比一元操作符高的操作符）：

```
-s($file) + 1024  # 可能错误；与-s($file + 1024)相同
(-s $file) + 1024  # 正确
```

需要说明，`-s/a/b/`不会完成一个“非”替换。不过，`-exp($foo)`仍按预期的方式工作，只有负号后的单个字母会被解释为文件测试。

文件权限操作符`-r`、`-R`、`-w`、`-W`、`-x`和`-X`的解释完全取决于文件的模式以及用户的用户ID和组ID。可能出于其他原因无法读、写或执行文件，如你的系统使用了存取控制列表（ACL，Access Control Lists），而你不在这个列表中<sup>注6</sup>，你就无法访问系统。另外要注意，对于超级用户，`-r`、`-R`、`-w`和`-W`总返回1，如果模式中设置了执行位，`-x`和`-X`也会返回1。因此，超级用户运行的脚本可能需要完成一个`stat`来确定文件的具体模式，或者假装不是超级用户（临时将UID设置为其他标识）。其他文件测试操作符不关心你是谁。任何人都可以对“常规”文件做这个测试：

注6： 不过，可以用`filetest pragma`覆盖内置语义。参见第29章。



```

while (<>) {
    chomp;
    next unless -f $_; # 忽略"特殊"文件
    ...
}

```

-T和-B开关的工作如下。首先在文件的第一个块中检查一些奇怪的字符，如控制码或者设置了高位的字节（看上去与UTF-8不同）。如果字节中三分之一以上都很奇怪，这就是一个二进制文件；否则，这是一个文本文件。另外，第一块中包含ASCII NUL (\0)的文件也认为是一个二进制文件。如果在一个文件句柄上使用-T或-B，则会检查当前输入（标准I/O或“stdio”）缓冲区，而不是文件的第一个块。对于空文件，-T和-B都返回true，另外测试一个文件句柄时，如果到达文件末尾（EOF，end-of-file），-T和-B也返回true。由于要完成-T测试必须读取文件，你肯定不希望在可能带来其他威胁的特殊文件上使用-T。所以，在大多数情况下，你会首先用-f测试，如下：

```
next unless -f $file && -T $file;
```

如果为stat、lstat或文件测试操作符指定了一个特殊的文件句柄，其中只包含一个下划线，就会使用前一个文件测试（或stat操作符）的stat结构，这样可以少做一个系统调用（声明了use filetest或者使用-t时不会这样做，要记住lstat和-l会把值留在对应符号链接的stat结构中，而不是真正的文件。类似的，在一个正常的stat之后，-l \_总为false）。

下面给出几个例子：

```

print "Can do.\n" if -r $a || -w _ || -x _;
stat($filename);
print "Readable\n" if -r _;
print "Writable\n" if -w _;
print "Executable\n" if -x _;
print "Setuid\n" if -u _;
print "Setgid\n" if -g _;
print "Sticky\n" if -k _;
print "Text\n" if -T _;
print "Binary\n" if -B _;

```

-M、-A和-C测试的文件年龄按脚本开始运行以来的天数返回（包括小数）。这个开始时间存储在特殊变量\$^T(\$BASETIME)中。因此，如果脚本启动之后文件有修改，就会得到一个负时间。需要说明，大多数时间值（约为86,399/86,400）都是小数，所以如果没有使用int函数，要测试文件年龄与一个整数是否相等往往是徒劳的。举例来讲：

```

next unless -M $file > .5;           # 文件年龄大于12小时
&newfile if -M $file < 0;           # 文件比进程新
&mailwarning if int(-A) == 90;       # 文件($_)在90天前访问过

```

要把脚本的开始时间重置为当前时间，可以如下指定：

```
$^T = time;
```

从v5.10开始，作为一个纯粹的语法糖，还可以“叠加”文件测试操作符，所以`-f -w -x $file`等价于`-x $file && -w _ && -f _`。

## 关系操作符

Perl有两类关系操作符。一类处理数值，另一类处理字符串值，如表3-5所示。

表3-5：关系操作符

数值	字符串	含义
>	gt	大于
>=	ge	大于等于
<	lt	小于
<=	le	小于等于

这些操作符返回1表示true，返回""表示false。需要说明，关系操作符不具有结合性，这说明`$a < $b < $c`将是一个语法错误。

如果没有本地化环境（locale）声明，字符串比较会基于字符串中各个字符的数值Unicode码点顺序来完成。如果有本地化环境声明，会使用本地化环境指定的排序顺序。原来这些基于本地化环境的排序机制与Unicode::Collate和Unicode::Collate::Locale模块提供的Unicode排序机制并不能很好地交互。最好使用模块而不是本地化环境。除了（单一的）ASCII外，码点顺序并不是字母顺序，所以Perl的字符串操作符只会在传统的ASCII数据上生成字母结果，而不能保证在任意文本都能生成字母结果。

## 相等操作符

表3-6所列的相等操作符与关系操作符很类似。

表3-6：相等操作符

数值	字符串	含义
==	eq	等于
!=	ne	不等于
<=>	cmp	比较，返回有符号的结果
~~	~~	智能匹配

等于和不等操作符会返回1表示true，返回""表示false（与关系操作符相同）。对于`<=>`和`cmp`操作符，如果左操作数小于右操作数，会返回-1，如果相等则返回0，如果左操作数大于右操作数，则返回+1。尽管相等操作符看起来与关系操作符很类似，不过相等操作符的优先级要低一级，所以`$a < $b <=> $c < $d`在语法上是合法的。

<=>操作符也称为“飞船”操作符（看过星球大战的人可能很清楚原因）。

~~操作符将在下一节介绍。

## 智能匹配操作符

二元操作符~~是v5.10.1中首次引入的<sup>注7</sup>，会在它的两个参数间完成一个“智能匹配”。它主要隐含地用在when构造中，不过并不是所有when子句都会调用这个智能匹配操作符。在Perl的所有操作符中，智能匹配操作符可算是独树一帜，因为它可以递归。

它的特别还体现在另一方面，Perl的所有其他操作符都会对其操作数提供一个上下文（通常是字符串或数值上下文），会把它的操作数自动转换到所提供的上下文。与之不同，智能匹配会从其操作数的实际类型推导出上下文，并使用这个类型信息来选择一个合适的比较机制。

~~操作符会“多态地”比较其操作数，根据操作数的实际类型（数值、字符串、数组、散列等）确定如何进行比较。与相等操作符类似（它们有相同的优先级），~~会返回1表示true，返回""表示false。类似于=~绑定操作符，这个操作符的右参数被认为是一个模式，可以接受或拒绝左参数。不过，“模式”这个概念的范围已经大大扩展，几乎任何值都可以作为模式，或者作为模式列表。

所以，通常最好把~~大声地读作“匹配”或“匹配任何”，因为左操作数会把自己提交给右操作数（或者右操作数的某一部分），由它接受或拒绝。

智能匹配的行为取决于它的参数是何种类型，如表3-7所示。表中第一行匹配类型将确定智能匹配的行为。由于具体工作由右操作数的类型来确定，然后才会由左操作数的类型确定，所以这个表按右操作数排序。

智能匹配会隐含地对所有未“祝福”的散列或数组引用解引用，所以HASH和ARRAY记录适用这些情况。已“祝福”的引用适用Object记录。涉及散列的智能匹配只考虑散列键，而不考虑散列值。

“匹配”列并不总能完全再现。例如，智能匹配操作符会尽可能短路，不过grep不会这样做。另外，标量上下文中grep会返回匹配个数，而~~只返回true或false。

与大多数操作符不同，智能匹配操作符知道要以特殊的方式处理undef：

```
my @array = (1, 2, 3, undef, 4, 5);
say "some elements undefined" if undef ~~ @array;
```

---

注7： 在一些特别的情况下，v5.10.0版本会有不同的表现，不过这没有关系，因为你现在至少在使用v5.14，没错吧？



一般认为各操作数在一个已修改的标量上下文中，这个修改是指：数组和散列变量会按引用传递给操作符，这会将它们隐式地解引用。各对中的两个元素是一样的：

```
my %hash = (red => 1, blue => 2, green => 3,
            orange => 4, yellow => 5, purple => 6,
            black => 7, grey => 8, white => 9);

my @array = qw(red blue green);

say "some array elements in hash keys" if @array ~~ %hash;
say "some array elements in hash keys" if \@array ~~ \%hash;

say "red in array" if "red" ~~ @array;
say "red in array" if "red" ~~ \@array;

say "some keys end in e" if /e$/ ~~ %hash;
say "some keys end in e" if /e$/ ~~ \%hash;
```

表3-7：智能匹配行为

左	右	描述	举例（不过在布尔上下文中计算）
<i>Any</i>	<code>undef</code>	检查 <i>Any</i> 是否未定义	<code>!defined Any</code>
<i>Any</i>	<code>Object</code>	在 <i>Object</i> 上调用重载 <code>~~</code> ，或者如果 <i>Object</i> 未重载则失败（die）	
<i>HASH</i>	<i>CODE</i>	子例程对所有 <i>HASH</i> 键返回true <sup>a</sup>	<code>!grep { !CODE-&gt;(\$_) } keys HASH</code>
<i>ARRAY</i>	<i>CODE</i>	子例程对所有 <i>ARRAY</i> 元素返回true <sup>a</sup>	<code>!grep { !CODE-&gt;(\$_) } ARRAY</code>
<i>Any</i>	<i>CODE</i>	向子例程传入 <i>Any</i> 时返回true	<code>CODE-&gt;(Any)</code>
<i>HASH1</i>	<i>HASH2</i>	两个 <i>HASH</i> 中所有相同的键	<code>keys HASH1 == grep { exists HASH2-&gt;{\$_} } keys HASH1</code>
<i>ARRAY</i>	<i>HASH</i>	<i>ARRAY</i> 元素作为 <i>HASH</i> 键	<code>grep { exists HASH-&gt;{\$_} } ARRAY</code>
<i>Regexp</i>	<i>HASH</i>	<i>HASH</i> 键与 <i>Regexp</i> 模式匹配	<code>grep { /Regexp/ } keys HASH</code>
<i>undef</i>	<i>HASH</i>	总为false（ <i>undef</i> 不可能是键）	<code>0 == 1</code>
<i>Any</i>	<i>HASH</i>	<i>HASH</i> 键是否存在	<code>exists HASH-&gt;{Any}</code>
<i>HASH</i>	<i>ARRAY</i>	<i>ARRAY</i> 元素作为 <i>HASH</i> 键	<code>grep { exists HASH-&gt;{\$_} } ARRAY</code>
<i>ARRAY1</i>	<i>ARRAY2</i>	在 <i>ARRAY1</i> 和 <i>ARRAY2</i> 的成对元素上递归处理 <sup>b</sup>	<code>(ARRAY1[0] ~~ ARRAY2[0]) &amp;&amp; (ARRAY1[1] ~~ ARRAY2[1]) &amp;&amp; ...</code>
<i>Regexp</i>	<i>ARRAY</i>	<i>ARRAY</i> 元素与 <i>Regexp</i> 模式匹配	<code>grep { /Regexp/ } ARRAY</code>
<i>undef</i>	<i>ARRAY</i>	<i>ARRAY</i> 中的 <i>undef</i>	<code>grep { !defined } ARRAY</code>
<i>Any</i>	<i>ARRAY</i>	智能匹配各个 <i>ARRAY</i> 元素 <sup>c</sup>	<code>grep { Any ~~ \$_ } ARRAY</code>

表3-7：智能匹配行为（续）

左	右	描述	举例（不过在布尔上下文中计算）
<i>HASH</i>	<i>Regexp</i>	<i>HASH</i> 键匹配 <i>Regexp</i>	<code>grep { /Regexp/ } keys HASH</code>
<i>ARRAY</i>	<i>Regexp</i>	数组元素匹配 <i>Regexp</i>	<code>grep { /Regexp/ } ARRAY</code>
<i>Any</i>	<i>Regexp</i>	模式匹配	<code>Any =~ /Regexp/</code>
<i>Object</i>	<i>Any</i>	在 <i>Object</i> 上调用重载 <code>~~</code> ，或者如果 <i>Object</i> 未重载则退为字符串或数值比较	
<i>Any</i>	<i>Num</i>	数值是否相等	<code>Any == Num</code>
<i>Num</i>	<i>numlike</i> <sup>d</sup>	数值是否相等	<code>Num == numlike</code>
<i>undef</i>	<i>Any</i>	检查是否未定义	<code>!defined(Any)</code>
<i>Any</i>	<i>Any</i>	字符串是否相等	<code>Any eq Any</code>

- a: 空散列或数组匹配。
- b: 也就是说，每个元素与另一个数组中相同索引的元素智能匹配。
- c: 如果找到一个循环引用，则退而判断引用是否相等。
- d: 可以是一个真正的数字，也可以是看上去像数字的字符串。

如果第一个数组中的各个元素与第二个数组中的相应元素能够递归地智能匹配（也就是说，“在第二个数组中”），这两个数组就能智能匹配：

```
my @little = qw(red blue green);
my @bigger = ("red", "blue", [ "orange", "green" ] );
if (@little ~~ @bigger) { # true!
    say "little is contained in bigger";
}
```

由于智能匹配操作符会在嵌套数组上递归处理，这仍会报告"red"在数组中：

```
my @array = qw(red blue green);
my $nested_array = [[[[[[ @array ]]]]]];
say "red in array" if "red" ~~ $nested_array;
```

如果两个数组相互智能匹配，这两个数组就互为深副本，如下例所示：

```
my @a = (0, 1, 2, [3, [4, 5], 6], 7);
my @b = (0, 1, 2, [3, [4, 5], 6], 7);

if (@a ~~ @b && @b ~~ @a) {
    say "a and b are deep copies of each other";
}
elsif (@a ~~ @b) {
    say "a smartmatches in b";
}
elsif (@b ~~ @a) {
```

```

        say "b smartmatches in a";
    }
    else {
        say "a and b don't smartmatch each other at all";
    }
}

```

运行这个脚本时，可以得到：

```
a and b are deep copies of each other
```

如果要设置**\$b[3] = 4**，就会报告“b smartmatches in a”，因为@a中相应的位置包含一个数组，其中（最终）会有一个4。

将一个散列与另一个散列智能匹配时，会报告这两个散列是否包含相同的键，而且不多也不少。这可以用来查看两个记录是否有相同的字段名，而不用关心这些字段的值。例如：

```

use v5.10;
sub make_dogtag {
    state $REQUIRED_FIELDS = { name=>1, rank=>1, serial_num=>1 };

    my ($class, $init_fields) = @_;

    die "Must supply (only) name, rank, and serial number"
        unless $init_fields ~~ $REQUIRED_FIELDS;

    ...
}

```

或者，如果允许有其他字段，但不要求必须有，可以使用`ARRAY ~~ HASH`：

```

use v5.10;
sub make_dogtag {
    state $REQUIRED_FIELDS = { name=>1, rank=>1, serial_num=>1 };

    my ($class, $init_fields) = @_;

    die "Must supply (at least) name, rank, and serial number"
        unless [keys %{$init_fields}] ~~ $REQUIRED_FIELDS;

    ...
}

```

智能匹配操作符最常用作when子句的隐式操作符。参见第4章的“given语句”一节。

## 对象的智能匹配

为了避免依赖于对象的底层表示，如果智能匹配的右操作数是未重载~~的对象，这会产生异常“Smartmatching a non-overloaded object breaks encapsulation”（智能匹配一个未重载对象会破坏封装）。这是因为，没有办法深入查看某个东西是否在对象中。对于没有重载~~的对象，以下做法都是不合法的：



```
%hash ~~ $object
42 ~~ $object
"fred" ~~ $object
```

不过，可以通过重载`~~`操作符来改变如何智能匹配一个对象。允许对通常的智能匹配语义进行扩展。对于确实重载了`~~`的对象，请参见第13章。

将对象用作左操作数是允许的，不过这样用处不大。智能匹配规则的优先级比重载要高，所以尽管左操作数中的对象提供了智能匹配重载，但会被忽略。如果左操作数是一个未重载对象，则会对`ref`操作符返回的结果完成一个字符串或数值比较。这说明：

```
$object ~~ X
```

并不会把`X`作为一个参数来调用重载方法，而是会参考前面给出的表，根据`X`的类型，可能调用也可能不调用重载。对于简单的字符串或数字，这等价于：

```
$object ~~ $number    ref($object) == $number
$object ~~ $string     ref($object) eq $string
```

例如，以下代码会报告“handle smells IOish”：

```
use IO::Handle;
my $fh = IO::Handle->new();
if ($fh ~~ /\bIO\b/) {
    say "handle smells IOish";
}
```

这是因为，它把`$fh`处理为一个类似“`IO::Handle=GLOBAL(0x8039e0)`”的字符串，然后对它完成模式匹配<sup>注8</sup>。

## 位操作符

类似C，Perl提供了位与（AND）、位或（OR）、异或（XOR）和非（NOT）操作符：`s`：`&`，`|`，`^`和前面介绍过的`~`。从这一章最前面的大表中你可能已经注意到，位与操作符比其他操作符的优先级都高，不过我们偷点懒，把这些操作符都放在这里讨论。

这些操作符对数值的处理与对字符串的处理有所不同（只有很少几种情况下Perl会关心二者的区别，这便是其中一种情况）。如果某个操作数为数字（或者曾被用作数字），两个操作符都将转换为整数，并在这两个整数上完成位操作。这些整数肯定至少为32位，不过有些机器上可能是64位。关键是对机器的体系结构有一个假定限制。可以用`bigint pragma`绕开这个限制。

如果两个操作数都是字符串（而且自设置以来没有当作数字使用过），这些操作符就会对

---

注8： 不过请不要这样做。将来很可能改为更接近Perl 6语义，即由右参数的类型确定左参数中的对象如何重载（字符串或数字）。所以现在要避免将对象放在左边。

两个字符串的相应位完成位操作。在这种情况下，不存在假定限制，因为字符串的大小没有假定限制。如果一个字符串比另一个字符串要长，较短的那个字符串会在末尾追加足够的0来补足差距。两个字符串中各个相应逻辑字符的位将相与、相或或者完成异或。

例如，如果将两个字符串相与：

```
"123.45" & "234.56"
```

会得到另一个字符串：

```
"020.44"
```

如果将一个字符串和一个数字相与：

```
"123.45" & 234.56
```

这个字符串首先转换为一个数字，这会得到：

```
123.45 & 234.56
```

然后再将这些数字转换为整数：

```
123 & 234
```

因此最后计算为106。需要注意，所有位串都为true（除非串中的结果是"0"）。这说明，如果你想查看某个字节是否非0，不要这样写：

```
if ( "fred" & "\x01\x02\x03\x04" ) { ... }
```

而要写为：

```
if ( ("fred" & "\x01\x02\x03\x04") =~ /[^\0]/ ) { ... }
```

## C风格的逻辑（短路）操作符

与C类似，Perl提供了&&（逻辑与）和||（逻辑或）操作符。Perl还提供了||的一个变种，即逻辑“定义或”操作符//。这些操作符都是从左向右计算（&&的优先级稍高于||或//），来计算语句的真值。这些操作符（见表3-8）也称为短路操作符，因为它们计算语句的真值时，会尽可能少地计算操作数。例如，如果一个&&操作符的左操作数是false，就不会计算右操作数，因为不论右操作数的值是什么，这个操作符的结果都是false。

表3-8：逻辑操作符

示例	名字	结果
<code>\$a &amp;&amp; \$b</code>	逻辑与	如果\$a为false则为\$a，否则为\$b
<code>\$a    \$b</code>	逻辑或	如果\$a为true则为\$a，否则为\$b

表3-8: 逻辑操作符 (续)

示例	名字	结果
<code>\$a // \$b</code>	定义或	如果\$a已定义则为\$a, 否则为\$b
<code>\$a and \$b</code>	低优先级与	如果\$a为false则为\$a, 否则为\$b
<code>\$a or \$b</code>	低优先级或	如果\$a为true则为\$a, 否则为\$b
<code>\$a xor \$b</code>	低优先级异或	\$a和\$b中只有一个为true时结果为true, 否则为false

这种短路不仅能节省时间, 还常常用来控制计算流程。例如, Perl程序中经常用到这样一个技巧:

```
open(FILE, "<", "somefile") || die "Can't open somefile: $!\n";
```

在这种情况下, Perl首先计算open函数。如果值为true (因为somefile被成功打开), 就没有必要执行die函数, 所以会将其跳过。可以从字面上读作“要么打开某个文件, 要么灭亡!”

对于通过返回undef来指示失败的函数, //操作符会很有用。

例如:

```
my $pid = fork() // die "Can't fork: $!";
if ($pid) {
    # 这里是父进程代码
    ...
    wait $pid;
} else {
    # 这里是子进程代码
    ...
    exit;
}
```

它还可以用于检测散列中没有的值。如果键不在一个散列中, 或者键有一个未定义的值, 它就会返回default:

```
$value = $hash{$key} // "DEFAULT";
```

&&和||操作符与C中的相应操作符有点区别, 它们并不返回0或1, 而是返回所计算的最后一个值。对于||, 其结果是你可以在一系列true标量值中选择第一个true值。所以, 查找用户主目录时可以采用一种可移植的方式:

```
my $home = $ENV{HOME}
           || $ENV{LOGDIR}
           || (getpwuid($<))[7]
           || die "You're homeless!\n";
```



另一方面，由于左参数总是在标量上下文中计算，所以不能使用`||`在两个集合间做出选择来完成赋值：

```
@a = @b || @c;           # 这样做的结果不对
@a = scalar(@b) || @c;    # 因为它的实际含义是这样
@a = @b ? @b : @c;        # 不过，这样是可以的
```

Perl还提供了较低优先级的逻辑与（`and`）和逻辑或（`or`）操作符，它们不需要对列表操作符加括号。有些人觉得这样更可读，不过也有些人认为这样不便于阅读。这些作为单词拼出的操作符也可以短路。完整的列表见表3-8。

## 范围操作符

取决于具体的上下文，`..` 范围操作符实际上是两个不同的操作符。

在标量上下文中，`..` 返回一个布尔值。这个操作符是双态的，就像一个电子触发器，可以模拟`sed`、`awk`和很多其他编辑器的行范围（逗号）操作符。每个标量`..`操作符会维护其自己的布尔状态。只要左操作数为`false`，它就是`false`。一旦左操作数为`true`，范围操作符就保持为`true`，直到右操作数为`true`，此后范围操作符又会变成`false`。在下一次计算之前，这些操作符不会变成`false`。它可能检查右操作数，并在右操作数变为`true`的同一次计算中也变为`false`（`awk`的范围操作符就是这样做的），不过它仍会返回一次`true`。如果不希望它在下一次计算前检查右操作数（`sed`的范围操作符采用这种做法），只需要使用3个点号（`...`）而不是2个<sup>注9</sup>。对于`..`和`...`，操作符处于`false`状态时不会计算右操作数，而在操作符处于`true`状态时不会计算左操作数。

返回的值可以是空串表示`false`，或者是一个序列号（从1计起）表示`true`。会为每一个遇到的范围重置这个序列号。范围中最后一个序列号将追加一个字符串“E0”，这不会影响它的数值，不过如果你想找出终点可以搜索这个字符串。要想找到起点，则要等待序列号大于1。如果标量`..`的任意一个操作数是一个数值直接量，这个操作数会隐式与`$. 变量`比较（这个变量中包含输入文件的当前行号）<sup>注10</sup>。

来看几个例子：

```
if (101 .. 200) { print }    # 打印101 .. 200
next line if 1 .. /^$/;      # 跳过消息的标题行
s/^/> / if /^$/ .. eof;      # 消息的引用体
```

在列表上下文中，`..`会返回包括从左值到右值的一个值列表。这对于编写`for (1..10)`循环以及对数组完成片段操作很有用：

---

注9： 不要把`...` 范围操作符与`...`省略语句混淆了，执行时这会产生一个“Unimplemented”异常。

注10： 从技术上讲，对于最后一个调用的句柄，它包含从该句柄最后一次关闭以来在这个句柄上调用`readline`操作符的次数。

```
for (101 .. 200) { print }    # 打印101102...199200
```

```
my @foo = getlist();  
@foo = @foo[0 .. $#foo];      # 开销巨大的no-op（无实际工作的操作）  
@foo = @foo[ -5 .. -1 ];      # 片段取最后5项
```

在当前实现中，在foreach循环中使用范围操作符作为表达式时，不会创建临时数组，不过如果有以下循环，老版本的Perl可能会耗费大量内存：

```
for (1 .. 1_000_000) {  
    # 代码  
}
```

如果左值大于右值，则返回一个空列表。（要以逆序生成列表，参见reverse操作符）。

如果操作数是字符串，范围操作符会利用前面讨论过的魔法自增算法。所以可以写为：

```
my @alphabet = ("A" .. "Z");
```

这会得到（现代英语）字母表的所有字母，或者写为：

```
my $hexdigit = (0 .. 9, "a" .. "f")[$num & 15];
```

这会得到一个十六进制数字，或者：

```
my @z2 = ("01" .. "31");  
print $z2[$mday];
```

可以得到有前导0的日期。还可以用：

```
my @combos = ("aa" .. "zz");
```

得到小写ASCII字母的所有两字母组合。不过，要当心以下代码：

```
my @bigcombos = ("aaaaaaa" .. "zzzzzzz");
```

因为这会需要大量内存。更确切地讲，这需要大量空间来存储8 031 810 176个标量。如果是这样，希望你有一个64位的机器，有数TB的内存，而且速度要很快。在这种情况下，可能更应该考虑采用一种迭代的方法。

如果指定的最后一个值不在魔法自增生成的序列中，这个序列会继续，直到下一个值比指定的最后一个值长。例如，“W” .. “M”会生成“W”，“X”，“Y”和“Z”，不过然后会结束，因为序列中的下一项“AA”比指定的“M”长。

如果指定的起始值不在魔法自增序列中（也就是说，一个与`/^[a-zA-Z]*[0-9]*\z/`匹配的非空字符串），则只返回起始值。所以下面的代码会返回一个alpha：

```
use charnames "greek";  
my @greek_small = ("\N{alpha}" .. "\N{omega}");
```

要得到小写的希腊字母，可以写作：

```
use charnames "greek";
my @greek_small = map { chr } (
    ord("\N{alpha}") .. ord("\N{omega}")
);
```

不过，这会选出一个多余的字母，因为rho和tau之间有两个不同的小写sigma，多余的那个是"\N{final sigma}"。一般都假设码点序列对应于字母表顺序。有关内容见第6章中“Unicode文本比较和排序”一节。

## 条件操作符

与C中一样，?:是唯一的三元操作符。通常把它称为条件操作符，因为它很类似一个if-then-else语句，只不过这是一个表达式而不是语句，可以安全地嵌入在其他表达式和函数调用中。作为一个三元操作符，它的两部分分隔了3个表达式：

```
COND ? THEN : ELSE
```

如果条件COND为true，只会计算THEN表达式，这个表达式的值将成为整个表达式的值。否则，只会计算ELSE表达式，它的值将成为整个表达式的值。

标量或列表上下文会向下传播到所选择的第二个或第三个参数（第一个参数总在标量上下文中，因为这是一个条件）。

```
my $a = $ok ? $b : $c;      # 得到一个标量
my @a = $ok ? @b : @c;      # 得到一个数组
my $a = $ok ? @b : @c;      # 得到一个数组中的元素个数
```

你会经常看到这个条件操作符嵌入在值列表中，用printf完成格式化，因为谁都不想只是为了切换两个相关的值而复制整个语句：

```
printf "I have %d camel%s.\n",
    $n, $n == 1 ? "" : "s";
```

很方便地，?:的优先级比逗号高，但是要低于大多数在它内部使用的操作符（如本例中的==），所以通常不用加括号。不过如果愿意，你也可以加上括号让代码更清晰。对于嵌套在其他条件操作符THEN部分中的条件操作符，建议你加上换行和缩进，就好像它们是正常的if语句一样：

```
$leapyear =
    $year % 4 == 0
        ? $year % 100 == 0
            ? $year % 400 == 0
                ? 1
                : 0
            : 1
```



```
: 0;
```

对于嵌套在之前条件操作符`ELSE`部分的条件操作符，可以做类似的处理：

```
$leapyear =
    $year % 4
    ? 0
    : $year % 100
    ? 1
    : $year % 400
    ? 0
    : 1;
```

不过，通常最好把所有`COND`和`THEN`部分垂直对齐：

```
$leapyear =
    $year % 4 ? 0 :
    $year % 100 ? 1 :
    $year % 400 ? 0 : 1;
```

将问号和冒号对齐会让很混乱的结构看起来也很清楚：

```
printf "Yes, I like my %s book!\n",
    $i18n eq "french" ? "chameau" :
    $i18n eq "german" ? "Kamel" :
    $i18n eq "japanese" ? "\x{99F1}\x{99DD}" :
    "camel"
```

利用`utf8 pragma`，甚至不用对Unicode字符转义：

```
use utf8;
printf "Yes, I like my %s book!\n",
    $i18n eq "french" ? "chameau" :
    $i18n eq "german" ? "Kamel" :
    $i18n eq "japanese" ? "骆驼" :
    "camel"
```

如果第2个和第3个参数都是合法的左值（说明可以为它们赋值），而且二者都是标量或者都是列表（否则Perl无法知道要对赋值右边提供哪个上下文），那么还可以为条件操作符赋值<sup>注11</sup>：

```
($a_or_b ? $a : $b) = $c; # 将$a或$b设置为$c的值
```

要记住，条件操作符比各种赋值操作符的优先级更高。通常正是你希望的（例如，见上面的`$leapyear`赋值），不过不使用括号就不能改变这一点。如果使用了嵌套赋值但是没有加括号，你可能会遇到麻烦，而且往往不会因此得到一个解析错误，因为条件操作符可以解析为一个左值。例如，你可能会写如下语句：

```
$a % 2 ? $a += 10 : $a += 2 # 错误
```

---

注11：这不一定能提高程序的可读性。不过可以在高难度Perl竞赛中用来创建很酷的表达式。

这会解析为：

```
(( $a % 2 ) ? ( $a += 10 ) : $a) += 2
```

## 赋值操作符

Perl能够识别C的赋值操作符，另外还提供了自己的一组赋值操作符。下面列出了这样一些赋值操作符：

=	**=	+=	*=	&=	<<=	&&=
		-=	/=	=	>>=	=
		.=	%=	^=		//=
			x=			

每个操作符都要求左边有一个目标左值（通常是一个变量或数组元素），右边是一个表达式。对于简单的赋值操作符：

```
TARGET = EXPR
```

*EXPR*的值存储在*TARGET*指定的变量或位置中。对于其他操作符，Perl会计算表达式：

```
TARGET OP= EXPR
```

就好像这个表达式写为：

```
TARGET = TARGET OP EXPR
```

这是一个很方便的基本原则，不过可能会在两方面有误解。首先，赋值操作符总是基于普通赋值的优先级完成解析，而不论*OP*本身的优先级是什么。其次，*TARGET*只计算一次。通常这并没有什么问题，除非有副作用，如自增：

```
$var[$a++] += $value;           # $a自增一次  
$var[$a++] = $var[$a++] + $value; # $a自增两次
```

与C中不同，赋值操作符会生成一个合法的左值。修改一个赋值就等价于完成赋值，然后将变量修改为所赋的值。这对于修改某个变量的副本会很有用，如下：

```
($tmp = $global) += $constant;
```

这等价于：

```
$tmp = $global + $constant;
```

类似的：

```
($a += 2) *= 3;
```

等价于：

```
$a += 2;
```

```
$a *= 3;
```

这还不算太有用，不过你会经常见到这样的用法：

```
(my $new = $old) =~ s/foo/bar/g;
```

在v5.14及以后版本中还可以这样写，使用/r修饰符返回修改后版本的一个副本，而不是作用于=~绑定的变量：

```
my $new = ($old =~ s/foo/bar/gr);  
my $new = $old =~ s/foo/bar/gr;
```

总之，赋值操作符的值将是变量的新值。由于赋值操作符从右向左结合，所以可以用来将多个变量赋为相同的值，如下：

```
$a = $b = $c = 0;
```

这会把0赋给\$c，然后这个赋值的结果（也是0）再赋给\$b，最后这个赋值结果（还是0）再赋给\$a。

列表赋值只能由普通的赋值操作符=完成。在列表上下文中，列表赋值会返回新值的列表，这与标量赋值类似。如第2章所述，在标量上下文中，列表赋值会返回赋值右边可用的值个数。这对于测试在不成功（或不再成功）的情况下返回一个空列表的函数很有用，如：

```
while (my ($key, $value) = each %gloss) { ... }  
next unless my ($dev, $ino, $mode) = stat $file;
```

## 逗号操作符

二元“,”是逗号操作符。在标量上下文中，它会在void上下文中计算其左参数，丢掉这个值，再在标量上下文中计算其右参数，并返回这个值。这与C的逗号操作符很类似。例如：

```
$a = (1, 3);
```

将3赋给\$a。不要把标量上下文用法与列表上下文用法混淆。在列表上下文中，逗号只是列表参数分隔符，将它的两个参数都插入到LIST中。它不会丢掉任何值。

例如，如果把前面这个例子改为：

```
@a = (1, 3);
```

这会构造一个两元素的列表，而

```
atan2(1, 3);
```



会调用函数atan2，并指定两个参数。

=>只是逗号操作符的一个同义字。它用于指示成对出现的参数。另外还强制将紧挨在它左边的标识符解释为一个字符串。这种自动引用只适用于标识符，而不适用于数值直接量。

## 列表操作符（右边）

列表操作符的右边包含列表操作符的所有参数，用逗号分隔，所以如果看右边，列表操作符的优先级低于逗号。一旦一个列表操作符开始处理用逗号分隔的参数，能让它停下来的只有终止整个表达式的token（如分号或语句修饰符）、终止当前子表达式的token（如右括号或中括号），或者低优先级逻辑操作符（接下来将要讨论）。

## 逻辑与、或、非和异或

作为比&&、||和!优先级低的逻辑操作符，Perl提供了and、or和not操作符。这些操作符的行为与相应的高优先级逻辑操作符是一样的，具体地，and和or会像对应的&&和||一样短路，所以它们不仅可以用于逻辑表达式，也可以用于控制流。

由于这些操作符的优先级比从C借用的那些操作符低得多，所以可以安全地用在列表操作符后面，而不用加括号：

```
unlink "alpha", "beta", "gamma"
    or gripe(), next LINE;
```

利用C风格的操作符，就必须写为：

```
unlink("alpha", "beta", "gamma")
    || (gripe(), next LINE);
```

不过不能直接把||的所有出现都替换为or。假设把以下语句：

```
$xyz = $x || $y || $z;
```

替换为

```
$xyz = $x or $y or $z; # 错误
```

这并不会做同样的事情！赋值的优先级比or高，但比||低，所以它会把\$x赋给\$xyz，然后完成逻辑或操作。要得到使用||时的效果，必须写为：

```
$xyz = ( $x or $y or $z );
```

这里要强调的是，不论你使用哪一种逻辑操作符，都必须好好学习优先级。建议你对可能让读者混淆的所有这些构造都加上括号，即使你自己不会弄错<sup>注12</sup>。

还有一个逻辑异或操作符（`xor`），这在C或Perl中没有对应的操作符，因为另外只有一个异或操作符（`^`），而它只作用于位（即只完成位异或操作）。`xor`操作符不能短路，因为两边都要计算。与`$a xor $b`最接近的可能是`!$a != !$b`。当然，也可以写为`!$a ^ !$b` 或者甚至`$a ? !$b : !!$b`。关键是`$a`和`$b`必须在布尔上下文中计算为`true`或`false`，而在没有帮助的情况下现有的位操作符不会提供布尔上下文。

## Perl中没有的C操作符

以下是C有但Perl没有的操作符：

### *unary &*

取地址（address-of）操作符。不过，Perl的`\`操作符（取引用）可以达到同样的目的：

```
$ref_to_var = \ $var;
```

但Perl引用比C指针安全得多。

### *unary \**

解除地址引用（dereference-address）操作符。因为Perl没有地址，所以不需要解除地址引用。不过，Perl有引用，所以Perl的变量前缀字符可以用作解引用操作符，还会指示类型：`$`、`@`、`%`和`&`。奇怪的是，确实有一个`*`解引用操作符，不过，由于`*`是指示类型团的印记，所以一般不会这样使用。

### *(TYPE)*

类型强制转换（`typecasting`）操作符。现在没人喜欢强制转换类型了。

---

注12：当然，除非你的本意就是想让读者学习优先级。我们真是同情这些读者。

# 语句和声明

Perl程序由一系列声明和语句组成。只要能放语句的地方都可以放声明，不过声明主要在编译时起作用。有些声明确实身兼两职，还可以作为正常的语句，不过大多数声明在运行时都可以看作是透明的。编译之后，主要语句序列只执行一次。

这一章中，我们先介绍语句，再讨论声明，不过需要指出，很多比较重要的声明会放在程序最前面。

与很多其他编程语言不同，Perl（默认地）不需要显式声明变量；变量在第一次使用时就会自然存在，而不论是否已经声明。不过，如果你愿意，也可以在第一次提到变量时进行声明，在变量名前面使用一个声明符（declarator），如`my`、`our`或`state`，这样一来，以后再次提到这个变量时，编译器就能确定你的变量名没有输错。

如果你想使用一个变量的值，但是还没有为这个变量赋过值，这种情况下，倘若把它用作一个数字，就会悄悄地处理为0，用作一个字符串时会处理为""（空串），当作一个逻辑值使用时则处理为false。如果你想对这种情况发出警告，则指示未定义的值被用作字符串或数字，`use warnings`声明会负责这个工作。

类似地，可以使用`use strict`声明来要求必须提前声明所有变量。如果使用了这个声明，所有未识别的变量名都将处理为语法错误。尽管`use v5.12`声明（或更高版本）会隐含地设置`strict`，不过我们建议使用`use v5.14`，这样本书中的例子才能正常编译和运行。关于这些声明的更多内容，请阅读这一章最后的“Pragmas”一节。

## 简单语句

简单语句就是一个表达式，计算这个表达式的目的只是为了得到它的副作用。每个简单语



句都必须以一个分号结束，除非这是代码块中的最后一条语句。在这种情况下，分号是可选的，Perl知道语句肯定已经结束，因为代码块已经结束。不过如果在一个多行代码块的最后，还是应该加一个分号，因为可能以后还会再加另外一行代码。

尽管类似`eval {}`、`do {}`和`sub {}`等操作符看上去都像是复合语句，但实际上它们并不是。没错，这些操作符允许包含多个语句，不过这不算是复合语句。从外部看，这些操作符只是表达式中的项，因此如果用作为一个语句中的最后一项，它们需要显式地增加一个分号。

所有简单语句后面都可以有一个修饰符（可选），如果有这样一个修饰符，则它要放在终止分号（或块末尾）前面。可能的修饰符包括：

```
if EXPR
unless EXPR
while EXPR
until EXPR
for LIST
when EXPR
```

`if`和`unless`修饰符的作用与英语中差不多：

```
$trash->take("out") if $you_love_me;
shutup() unless $you_want_me_to_leave;
```

`while`和`until`修饰符会反复计算。可以想见，对于`while`修饰符，只要其表达式保持为`true`，`while`修饰符就会一直执行表达式，不过，对于`until`修饰符，只有当其表达式保持为`false`时它才会一直执行：

```
$expression++ while -e "$file$expression";
kiss("me") until $I_die;
```

`for`修饰符（如果你不拒绝多敲几个键，也可以拼为`foreach`）会对其中`LIST`的每一个元素计算一次，每次计算时，用`$_`作为当前元素的别名：

```
s/java/perl/ for @resumes;
say "field: $_" foreach split /:/, $dataline;
```

`while`和`until`修饰符采用通常的`while-loop`语义（先计算条件），不过应用到一个`do BLOCK`（参见第27章）时是个例外，在这种情况下，计算条件之前会先执行一次代码块。这就允许你写类似下面的循环：

```
do {
    $line = <STDIN>;
    ...
} until $line eq ".\n";
```

需要指出，后面将要介绍的循环控制操作符在这个构造中无法使用，因为修饰符不接受循环标签。可以用一个额外的块包围这个构造让它提前终止，或者在内部加一个块来提前迭

代，见本章后面“裸块作为循环”一节介绍。或者也可以写一个包含多个循环控制的真正的循环（见下一节）。

`when`修饰符是一个处于试验阶段的特性，如果声明了`use v5.14`（或更高版本）就可以得到这个特性。它的模式匹配语义与`when`语句的语义是等价的，可以参考本章后面的“`when`语句和修饰符”一节。

## 复合语句

一个作用域<sup>注1</sup>中的一个语句序列称为一个代码块（*block*）。有时这个作用域是整个文件，如用`require`加载的一个文件或包含主程序的文件。有时这个作用域是用`eval`计算的一个字符串。不过，一般来讲，代码块都用大括号`{}`包围。谈到作用域时，我们就是指这3种作用域。如果是指带大括号的代码块，我们会用`BLOCK`来表示。

复合语句由表达式和`BLOCK`构造。表达式由项和操作符构成。在我们的语法描述中，将用`EXPR`来指示在这里可以使用标量表达式。要指示在列表上下文中计算的一个表达式，我们会用`LIST`表示。

可以用下面的语句来控制有条件地执行`BLOCK`以及重复执行`BLOCK`（`LABEL`部分可选）。

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ...
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

unless (EXPR) BLOCK
unless (EXPR) BLOCK else BLOCK
unless (EXPR) BLOCK elsif (EXPR) BLOCK ...
unless (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK

given (EXPR) BLOCK
LABEL while (EXPR) BLOCK
LABEL while (EXPR) BLOCK continue BLOCK
LABEL until (EXPR) BLOCK
LABEL until (EXPR) BLOCK continue BLOCK
LABEL for (EXPR; EXPR; EXPR) BLOCK
LABEL foreach (LIST) BLOCK
LABEL foreach (LIST) BLOCK continue BLOCK
LABEL foreach VAR (LIST) BLOCK
LABEL foreach VAR (LIST) BLOCK continue BLOCK
LABEL BLOCK
LABEL BLOCK continue BLOCK
```

需要说明，与C和Java中不同，这些是按`BLOCK`定义的，而不是按语句来定义。这说明大

---

注1：作用域和命名空间在第2章的“名字”一节中做过介绍。

括号是必不可少的，不允许有“悬空”的语句。如果想写不加大括号的条件语句，则有很多方法。下面的语句有相同的作用：

```
unless (open(F00, '<', $foo)) { die "Can't open $foo: $!" }
if (!open(F00, '<', $foo))    { die "Can't open $foo: $!" }

die "Can't open $foo: $!"    unless open(F00, '<', $foo);
die "Can't open $foo: $!"    if !open(F00, '<', $foo);

open(F00, '<', $foo)          || die "Can't open $foo: $!";
open(F00, '<', $foo)          or die "Can't open $foo: $!";
```

大多数情况下，我们更喜欢最后两种方法。与其他形式相比，这两种看上去不那么混乱，特别是“or die”版本。对于||形式，我们要习惯一丝不苟地使用小括号，不过对于or版本，如果忘了加小括号也没有太大关系。

不过我们更喜欢最后两个版本的主要原因是，它们将语句最重要的部分放在这一行的最前面，这样你会最先看到它。错误处理被放在一边，你不用特别注意这一部分，除非你确实很关注错误处理<sup>注2</sup>。如果每次都利用制表符（tab）把所有“or die”检查放在右边的同一列上，这样将更容易阅读：

```
chdir($dir)          or die "chdir $dir: $!";
open(F00, '<', $file)  or die "open $file: $!";
@lines = <F00>        or die "$file is empty?";
close(F00)           or die "close $file: $!";
```

## if和unless语句

if语句很直接。由于BLOCK总是用大括号包围，所以要确定一个else或elsif跟着哪一个if，这绝对不会有歧义。在任何给定的if/elsif/else BLOCK序列中，只会执行第一个条件计算为true的BLOCK。如果所有条件都不为true，则执行else BLOCK（如果有）。在一系列elsif末尾放一个else通常是个好主意，这样可以避免遗漏情况。

如果使用unless取代if，测试逻辑会反过来。也就是说：

```
unless ($x == 1) ...
```

等价于：

```
if ($x != 1) ...
```

甚至等价于（这不容易看明白）：

```
if (!($x == 1)) ...
```

---

注2：（就像这个脚注）。



对于控制条件中声明的变量，它的作用域会从其声明延伸到条件语句的其余部分，包括所有elsif和最后的else子句（如果有），但是不会超出这个范围：

```
if ((my $color = <STDIN>) =~ /red/i) {
    $value = 0xFF0000;
}
elsif ($color =~ /green/i) {
    $value = 0x00FF00;
}
elsif ($color =~ /blue/i) {
    $value = 0x0000FF;
}
else {
    warn "unknown RGB component '$color', using black instead\n";
    $value = 0x000000;
}
```

在else之后，\$color变量已经出了其作用域。如果希望作用域进一步延伸，就应当在条件语句前面声明这个变量。

## given语句

在前面的例子中，我们一直在讨论\$color。语言学家把这称作一个主题（topic）。v5.10和更高版本中提供了if结构的一个替代结构，即given函数，从语言学的角度讲，这就相当于一个主题工具（topicalizer）。它的做法是将\$\_设置为当前主题。然后可以使用when语句检查这个主题的各个不同值或模式。

使用Perl的v5.10及以后版本时，就会启用这个特性：

```
use v5.12;                                # 至少v5.12，加载默认特性
```

如果要特别请求“switch”特性，则需要：

```
use feature "switch";                    # 只得到switch特性
```

这两个声明都会向Perl语言增加一些新的关键字：given、when、break、continue和default。可以使用这个新特性重写前面的例子，以下给出了这样一种方法：

```
use v5.10;

my $value;
given (<STDIN>) {
    when (/red/i)    { $value = 0xFF0000 }
    when (/green/i) { $value = 0x00FF00 }
    when (/blue/i)  { $value = 0x0000FF; }
    default {
        warn "unknown RGB component '$_', using black instead\n";
        $value = 0x000000;
    }
}
```

```
}
```

实际上，在v5.10中必须这样写，因为given不能返回值。而在v5.14或更高版本中，你可以返回值，利用when的语句修饰符形式，甚至可以这样写：

```
use v5.14;

my $value = do {
    given (<STDIN>) {
        0xFF0000 when /red/i;
        0x00FF00 when /green/i;
        0x0000FF when /blue/i;
        warn "unknown RGB component '$_', using black instead\n";
        0x000000;
    }
};
```

given和when的参数都在标量上下文中计算；given将其参数绑定到\$\_变量，用来设置其BLOCK的主题。when通过查看参数的类型，使用其参数选择希望完成哪一种模式匹配。when的语义是智能匹配的一个超集。如果参数看起来是一个布尔表达式，则会直接计算。如果不是，则将把它传递到智能匹配操作符，解释为\$\_ ~~ EXPR。这看起来很复杂，不过实际上并不复杂，因为绝大多数switch都采用这种形式：

```
use v5.14;

my $n = somefunc();

given ($n) {
    when (0) { say "zero" }
    when (1) { say "one" }
    when ([3..7]) { say "many" }
    when (/^\d+$/) { say "lots" }
    default { say "unwholesome" }
}
```

换句话说，when参数通常会调用智能匹配（或者你可以假装在调用智能匹配）。

下面是given的一个更长一点的例子：

```
use feature ":5.10";

given ($n) {

    # 如果!defined($n)则匹配
    when (undef) {
        say '$n is undefined';
    }

    # 如果$n eq "foo"则匹配
    when ("foo") {
        say '$n is the string "foo"';
    }
}
```

```

# 如果 $n ~~ [1,3,5,7,9]则匹配
when ([1,3,5,7,9]) {
    say '$n is an odd digit';
    continue; # 直通!!
}

# 如果 $n < 100则匹配
when ($_ < 100) {
    say '$n is numerically less than 100';
}

# 如果complicated_check($n)则匹配
when (\&complicated_check) {
    say 'a complicated check for $n is true';
}

# 如果其他情况都不匹配
default {
    die q(I don't know what to do with $n);
}
}

```

`given(EXPR)`将`EXPR`的值赋给`$_`的一个词法作用域副本，而不是像没有`my`声明的`foreach`那样赋给一个动态作用域别名。这使它非常类似于一个`do`块：

```
do { my $_ = EXPR; ... }
```

不过成功的`when`（或任何显式的`break`）知道如何跳出这个块。由于它是词法作用域变量，所以可以使用`given`局部化`$_`的一个动态值，就像用老式的`foreach`一样<sup>注3</sup>。

可以使用`break`关键字跳出包围的`given`块。每个`when`块都隐含地用一个`break`结束。

可以使用`continue`关键字从一个分支跳出，直接前进到下一个语句的开始，这可能是另一个`when`（也可能不是）：

```

given($foo) {
    when (/x/) { say '$foo contains an x'; continue }
    say "I always get here.";
    when (/y/) { say '$foo contains a y' }
    default { say '$foo does not contain a y' }
}

```

`given`语句同时是一个合法的表达式时（例如，这是块中的最后一个语句），它会计算为：

- 一旦遇到一个显式的`break`，则计算为一个空列表。
- 成功的`when/default`子句（如果有）最后计算的表达式的值。

---

注3： 这一点被理解为一个特性，因为只要有两段不同的代码都在处理当前主题，动态作用域的`$_`将很容易出错。



- 如果没有条件为true，则为given块最后计算的表达式的值。

最后一个表达式要在为given块提供的上下文中计算。

需要说明，与if和unless不同，失败的when语句总是计算为一个空列表。

```
my $price = do {
    given ($item) {
        when (["pear", "apple"]) { 1 }
        break when "vote"; # 我的选票不容贿赂
        1e10 when /Mona Lisa/;
        "unknown";
    }
};
```

需要注意，这里必须使用do块，因为given只识别为一个语句，而它在赋值后是不合法的（在将来的版本中可能会让do块为可选）。

## when语句和修饰符

given的非凡能力主要来自于各个数据类型隐含的隐式智能匹配。默认地，when(EXPR)会处理为\$\_的一个隐式智能匹配；也就是 `$_ ~~ EXPR`（有关智能匹配的更多细节请参见第3章）。不过，如果when的EXPR参数属于以下所列的10种例外形式，则会直接计算，来得到一个布尔结果，不会发生智能匹配：

1. 用户自定义的子例程调用或方法调用。
2. 形如/REGEX/、\$foo =~ /REGEX/或\$foo =~ EXPR的正则表达式匹配。
3. 使用显式~~操作符的智能匹配，如EXPR ~~ EXPR（例如，如果需要取消when的内置智能匹配的默认多态处理，可以对\$\_使用一个显式的智能匹配）。
4. 返回布尔结果的关系操作符（如\$\_ < 10或\$x eq "abc"）。这包括6个数值比较操作符（<，>，<=，>=，==，和!=）以及6个字符串比较操作符（lt，gt，le，ge，eq和ne）。
5. 3个内置函数：defined、exists和eof。
6. 取非的表达式，可以是!EXPR或not(EXPR)，或者是逻辑异或EXPR1 xor EXPR2（这不包括按位取非和位异或操作[~ 和^]）。取非的正则表达式也属此类，而不论采用什么写法，可以是!/REGEX/、\$foo !~ /REGEX/或\$foo !~ EXPR。
7. 文件测试操作符（除-s、-M、-A和-C以外，因为这些操作符会返回数字，而不是布尔结果）。
8. ..和...触发器操作符（需要注意，...中缀操作符与...省略语句完全不同，只有当希望得到一个语句时才会识别省略语句）。

在以上8种情况中，*EXPR*的值直接用作一个布尔值，而不会完成智能匹配。可以把`when`看作一个智能的智能匹配（`smartsmartmatch`）<sup>注4</sup>。为了让它更为智能，Perl对逻辑操作符（即“与”和“或”）的操作数递归地应用这些测试，来确定是否使用智能匹配，如下所示：

1. 对于`EXPR1&&EXPR2`或`EXPR1 and EXPR2`，会对`EXPR1`和`EXPR2`递归地应用这个测试。只有当两个操作数都通过测试时才会把这个表达式处理为一个布尔值。否则将使用智能匹配。
2. 对于`EXPR1||EXPR2`或`EXPR1 or EXPR2`，只会对`EXPR1`递归地应用这个测试（例如，它本身可能是一个更高优先级的AND操作符，因此需要遵循前一条规则），而不对`EXPR2`应用测试。如果`EXPR1`要使用智能匹配，那么`EXPR2`也要使用智能匹配，而不论`EXPR2`包含什么内容。不过，如果`EXPR1`不使用智能匹配，那么第二个参数也不会使用。这与前面介绍的&&情况完全不同，所以要当心（需要说明，由于//操作符左边隐含有`defined`函数，所以`EXPR1//EXPR2`总被认为是布尔值）。

以上这些规则使问题看上去很复杂，但实际上并没有那么复杂。之所以有这些规则是因为Perl 5没有内置的布尔类型<sup>注5</sup>。这些规则的目标就是让你做到你想做的事情。例如：

```
when (/^\d+$/ && $_ < 75) { ... }
```

会被处理为一个布尔匹配，因为这些规则会把合取（*conjunction*）的两边都识别为布尔匹配。

另外：

```
when ([qw(foo bar)] && /baz/) { ... }
```

会使用智能匹配，因为只是第二个操作数看起来像是一个布尔值。第一个不是布尔值，所以这里智能匹配“取得胜利”，或者也许根本没有获胜方，除非你希望对右边的结果完成智能匹配，这将是1或""。这些`given`可能都不会提供。

要记住，顺序对于析取（*disjunction*）非常重要。如果有：

```
when ([qw(foo bar)] || /^baz/) { ... }
```

它会根据第一个操作数来使用智能匹配。不过：

```
when (/^baz/ || [qw(foo bar)]) { ... }
```

---

注4： 可以把布尔运算看作是智能匹配的一部分，这也很有用，因为在Perl 6中确实是这样，你可能需要不时地转换思路。

注5： 至少现在还没有。Perl将来的版本可能会增加一个布尔类型，如果是这样，这些复杂的规则就会很自然地放弃智能匹配。

首先是（布尔）表达式，这会强制将两个操作数都处理为布尔值，同样的，这里没有获胜方，因为第二个参数（一个数组引用）总是true，所以结果不会像你期望的那样。

基于常量的布尔操作符还会进一步优化。不要这样写：

```
when ("foo" or "bar") { ... }
```

这会优化为"foo"，所以永远不会考虑"bar"（尽管规则称要在"foo"上使用智能匹配）。如果要做类似这样的选择，完全可以使用数组引用，因为这会引发智能匹配，它有自己的“匹配任何一个”语义：

```
when (['foo', 'bar']) { ... }
```

对于有多个“标签”的情况就要这样写，因为在Perl中没有与C中fall-through语义对应的特性（在C中，fall-through是指switch语句中代码段与下一个case语句之间没有break）。

default就相当于when(1 == 1)，也就是说，它总会匹配。由于各个分支（情况）是按顺序计算的，所以它会最后计算；与when类似，它会有一个隐式的break，所以不会到达后面的代码。

作为对智能匹配语义的一个补充，如果使用一个直接量数组或散列作为given的参数，它会转换为一个引用，从而不会丢失任何信息。所以，例如given(@foo)与given(\@foo)就完全相同。如果确实想匹配@foo的长度，则需要写为given(scalar @foo)。

我们认为given和when的一些不好的方面还有待进一步试验，不过可以相信，在实际中，你的大多数switch语句都会基于简单的字符串或数字匹配，所以通常能得到你预想的结果。

## 循环语句

所有循环语句在其正式语法中都有一个可选的LABEL（可以在任何语句上加标签，不过标签对于循环来说有特殊的含义<sup>注6</sup>）。如果有标签，则它包括一个标识符，后面跟着一个冒号。通常标签会使用大写，一方面看起来比较明显，另外也可以避免与保留字冲突（如果你使用的标签本身已经有含义（如or或open），则Perl不会弄混，不过你的读者可能会不清楚）。

## while和until语句

只要EXPR为true，while语句就会反复执行。如果while放在until后面，这个测试的逻辑就要反过来；也就是说，只有当EXPR保持为false时才会执行代码块。不过，仍会在第一次迭代之前测试条件。

---

注6： 在v5.14之前，不能在package语句上加标签。



`while`或`until`语句可以有一个额外的代码块（可选）：`continue`。每次继续迭代时都会执行这个代码块，这有两种可能，一种可能是已经退出第一个代码块，另一种可能是使用了一个显式的`next`（这是一个循环控制操作符，要求进入下一次迭代）。`continue`块在实际中使用不算多，不过我们在这里用到了`continue`块，以便下一节严格地定义三部分循环。

与稍后将要介绍的`foreach`循环不同，`while`循环没有正式的“循环变量”<sup>注7</sup>。不过可以显式地声明变量。对于在`while`或`until`语句的测试条件中声明的变量，它们只在该测试控制的代码块（可能多个）中可见。这不属于其外围作用域。例如：

```
while (my $line = <STDIN>) {  
    $line = lc $line;  
}  
continue {  
    print $line; # 仍可见  
}  
# $line现在已经超出了作用域
```

在这里，`$line`的作用域从控制表达式中的声明开始，一直到循环构造的最后，包括`continue`块，但到此为止，不会超出。如果希望作用域继续延伸，则要在循环之前声明这个变量。

## 三部分循环

三部分循环<sup>注8</sup>的括号中有3个表达式，用分号分隔。这3个表达式分别解释为循环的初始化、条件和重新初始化。它们两边的括号和它们之间的两个分号是必不可少的，不过表达式本身可选。如果省略了初始化和重新初始化部分，它们什么也不做。如果忽略了条件，则认为有一个`true`值（初始化和重新初始化部分的值不重要，因为计算这两个表达式的目的只是为了得到其副作用）。

三部分循环可以根据相应的`while`循环来定义，只是要调整3个表达式的位置。如果有：

```
LABEL:  
    for (my $i = 1; $i<= 10; $i++) {  
        ...  
    }
```

这在内部会重新组织为：

---

注7： 这样带来的一个后果是，`while`不会在其测试条件中隐含地本地化任何变量。`while`循环与隐式地了解全局变量（如`$_`）的一些操作符结合使用时，会有一些“有趣的”结果。具体的，请参见第2章“行输入（尖角）操作符”一节，可以了解某些`while`循环中如何对全局变量`$_`隐式赋值，其中还提供了一些例子介绍如何解决这个问题。

注8： 也称为`for`循环，不过这有些混淆，因为Perl还有一些并非三部分循环的`for`循环，所以我们没有使用这个术语，而是称之为三部分循环。

```

{
    my $i = 1;
    LABEL:
    while ($i <= 10) {
        ...
    }
    continue {
        $i++;
    }
}

```

只不过实际上并没有一个外部块；我们在这里加了一个外部块只是为了说明my的作用域如何限定。

如果想同时迭代处理两个变量，只需要用逗号分隔这些并行表达式：

```

my $i;
my $bit;
for ($i = 0, $bit = 0; $i < 32; $i++, $bit <= 1) {
    say "Bit $i is set" if $mask & $bit;
}
# $i和$bit中的值在循环之后依然保留

```

或者，可以声明这些变量只在循环内部可见：

```

for (my ($i, $bit) = (0, 1); $i < 32; $i++, $bit <= 1) {
    say "Bit $i is set" if $mask & $bit;
}
# 循环中的$i和$bit现在超出了作用域

```

除了正常地循环处理循环索引，三部分循环还有很多其他有意思的应用。它甚至不需要一个显式的循环变量。下面给出一个例子，可以避免在交互式文件描述符上显式测试是否达到文件末尾（end-of-file）时可能遇到的问题（这可能会导致你的程序挂起）：

```

$on_a_tty = -t STDIN && -t STDOUT;
sub prompt { print "yes? " if $on_a_tty }
for ( prompt(); <STDIN>; prompt() ) {
    # 做些处理
}

```

三部分循环的另一种传统用法是“无限循环”。由于这3个表达式都是可选的，而且默认条件为true，所以如果写为

```

for (;;) {
    ...
}

```

这就相当于：

```

while (1) {
    ...
}

```

如果无限循环让你有些担心，可以告诉你，在循环的任意位置利用一个显式的循环控制操作符（如`last`）就能跳出循环。当然，如果你想编写代码来控制一个核导弹，可能并不需要一个显式的循环退出。循环会在适当的时刻自动终止<sup>注9</sup>。

## foreach循环

这个循环会迭代处理一个值列表，每次迭代时将控制变量(*VAR*) 设置为列表的各个后继元素：

```
for my VAR (LIST) {  
    ...  
}
```

如果忽略“`my VAR`”，则会使用全局变量`$_`。可以忽略`my`，不过只有在关闭`use strict`时才允许这样做，所以最好不要忽略`my`。

出于历史原因，`foreach`关键字是`for`关键字的同义字，所以可以交替地使用`for`和`foreach`，在给定情况下可以使用你认为更可读的一个。我们倾向于`for`，因为我们比较懒，另外也因为它确实更可读，特别是结合`my`时[不用担心，Perl能很容易地区分`for (@ARGV)`和`for ($i=0; $i< $#ARGV; $i++)`，因为后者包含分号]。下面给出几个例子：

```
$sum = 0;  
for my $value (@array) { $sum += $value }  
  
for my $count (10,9,8,7,6,5,4,3,2,1,"BOOM") { # 倒数  
    say $count;  
    sleep(1);  
}  
  
for (reverse "BOOM", 1 .. 10) {          # 完成同样的工作  
    say;  
    sleep(1);  
}  
  
for my $field (split /\:/, $data) {      # 任何LIST表达式  
    say "Field contains: '$field'";  
}  
  
for my $key (sort keys %hash) {  
    say "$key => $hash{$key}";  
}
```

最后一个例子是按排序顺序打印散列值的一种规范方法。更详细的例子参见第27章中关于`keys`和`sort`的介绍。

无法得到你在列表中的位置。要比较相邻的元素，可以把前一个元素记在一个变量里，不过有时必须利用下标写一个三部分循环。这正是提供两种不同循环的原因。

注9： 也就是说，循环退出会自动发生。



如果`LIST`包含可赋值的元素（一般来讲，这是指变量，而不是枚举常量），可以通过修改循环中的`VAR`来修改`LIST`中的各个变量。这是因为，循环变量会成为要循环处理的列表中各项的一个隐式别名。不仅可以在原地修改一个数组，还可以修改一个列表中的多个数组和散列：

```
for my $pay (@salaries) {                # 涨8%
    $pay *= 1.08;
}

for (@christmas, @easter) {              # 修改菜单
    s/ham/turkey/;
}
s/ham/turkey/ for @christmas, @easter; # 完成同样的工作

for ($scalar, @array, values %hash) {
    s/^\s+//;                            # 删除前导空白符
    s/\s+$//;                            # 删除结尾空白符
}
```

循环变量只在这个循环的动态或词法作用域中是有效的，如果这个变量先前是用`my`声明的，它将隐含地作为词法作用域变量。这样一来，在该变量的词法作用域之外定义的函数将不能看到这个变量，即使函数是从循环内部调用的，也无法看到这个变量。不过，如果作用域中没有词法声明，这就允许循环内部调用的函数访问这个变量。不论哪一种情况，这个局部变量会在循环退出时自动恢复它在循环之前的值。

如果你愿意，可以显式地声明要使用哪一种变量（词法作用域变量还是全局变量）。这样能让维护代码的人更容易地知道到底在做什么；否则，他们需要再来查找外围作用域，查看之前是否有一个声明，从而得出这是哪一种变量：

```
for my $i (1 .. 10) { ... }              # $i是词法作用域变量
for our $Tick (1 .. 10) { ... }          # $Tick是全局变量
```

如果循环变量有相应的声明，我们更倾向于使用拼写更简短的`for`而不是`foreach`，因为`for`在英语中更容易理解。

C或Java程序员最初可能认为用Perl编写一个特定算法时会是这样：

```
for ($i = 0; $i < @ary1; $i++) {
    for ($j = 0; $j < @ary2; $j++) {
        if ($ary1[$i] > $ary2[$j]) {
            last;                # 不能进入外层循环
        }
        $ary1[$i] += $ary2[$j];
    }
    # last会把我们带到这里
}
```

不过，经验丰富的Perl程序员可能会这样写：

```

WID: for my $this (@ary1) {
    JET: for my $that (@ary2) {
        next WID if $this > $that;
        $this += $that;
    }
}

```

看到了吧，采用符合Perl语言习惯的做法是不是容易多了？这个程序更简洁、更安全而且速度更快。更简洁的原因是，这里代码比较少。说它更安全是因为如果以后在内循环和外循环之间增加代码，不会意外地执行新代码，因为next（下面将要解释）会显式地迭代外循环，而不只是跳出内循环。另外相对于与它等效的三部分循环来讲，这种做法也更快，因为它可以直接访问元素，而不用通过下标。

不过你可以选择你喜欢的方式，毕竟，TMTOWTDI（不只一种方法做一件事）。

类似于while语句，foreach语句也可以有一个continue块。这允许你在每次循环迭代的最后执行一段代码，而不论你是正常地到达那里，还是通过一个next结束当前迭代。

现在我们终于要说到它了，下面就来讨论next。

## 循环控制

之前已经提到，可以在循环中放一个LABEL，为它指定一个名字。循环的LABEL会为循环控制操作符next、last和redo标识这个循环。LABEL是对整个循环命名，而不只是最上层循环。因此，引用循环的循环控制操作符并不真正“前往”循环标签本身。对计算机而言，这个标签甚至可以放在循环的最后。不过，出于某种原因，人们总喜欢把标签放在最前面。

通常会为每次迭代处理的项而对循环命名。它与循环控制操作符可以很好地交互，这些操作符的设计原则是：结合适当的标签和语句修饰符使用时，读起来应该像英语一样。原先的循环是按行处理的，所以原先的循环标签为LINE:，原先的循环控制操作符如下：

```
next LINE if /^#/;    # 丢掉注释
```

循环控制操作符的语法如下：

```

last LABEL
next LABEL
redo LABEL

```

LABEL是可选的；如果没有LABEL，操作符会指示最内层外围循环。不过，如果想跳过多层，就必须使用一个LABEL为你想影响的循环命名。这个LABEL不必在词法作用域中（尽管它可能应该在）。实际上，LABEL可以在动态作用域中的任何地方。如果这要求你跳出一个eval或子例程，Perl会（根据需要）发出一个警告。

就像函数中可以有多个return操作符一样，循环中也可以根据需要有多个循环控制操作符。不要认为这样不好或者不合理。在结构化编程发展初期，一些人坚持认为循环和子例程只能有一个入口和一个出口。这种单入口的想法仍然很好，不过单出口的概念让人们写出了大量不自然的代码。程序中很大一部分都是在遍历决策树。很自然地，决策树从一个主干开始，不过最后会有很多个叶子。编写代码时，如果有多个循环出口（和函数返回），这对于你要解决的问题会更自然。如果为变量声明了合理的作用域，会在适当的时刻自动地进行清理，而不论你以何种方式退出代码块。

last操作符会立即退出当前循环。即使有continue块也不会执行。下面的例子会在遇到第一个空行时退出循环：

```
LINE: while (<STDIN>) {
    last LINE if /^$/; # 处理完邮件首部就退出
    ...
}
```

next操作符会跳过当前循环迭代的其余部分，开始一个新的迭代。如果循环中有一个continue子句，会在重新计算条件之前执行，就像三部分for循环的第三个分量一样。因此，continue子句可以用来递增循环变量（即使某次循环迭代被next中断）：

```
LINE: while (<STDIN>) {
    next LINE if /^#/;      # 跳过注释
    next LINE if /^$/;      # 跳过空行
    ...
} continue {
    $count++;
}
```

redo操作符会重新开始循环块，而不会再次计算条件。即使有continue块也不会执行。如果程序想要了解刚才输入了什么，通常会使用这个操作符。假设你在处理一个文件，这个文件有些行的末尾有一个反斜线，表示下一行仍是同一个记录。为此可以使用redo，如下所示：

```
while (<>) {
    chomp;
    if (s/\\$//) {
        $_ .= <>;
        redo unless eof; # 读文件时不能超过各个文件的eof
    }
    # 现在处理$_
}
```

这是常用的Perl简写版本，与它对应还有更明确（也更冗长）的版本：

```
LINE: while (defined($line = <ARGV>)) {
    chomp($line);
    if ($line =~ s/\\$//) {
        $line .= <ARGV>;
    }
}
```



```

        redo LINE unless eof(ARGV);
    }
    # 现在处理$line
}

```

这是一个真实程序的例子，在这里3个循环控制操作符都用到了。尽管这种解析命令行参数的策略现在已经不常用了，因为Perl已经打包提供了Getopt::\*模块<sup>注10</sup>，但这种方法仍能很好地展示如何在命名嵌套循环上使用循环控制操作符：

```

ARG: while (@ARGV && $ARGV[0] =~ s/^(?=.)//) {
    OPT: for (shift @ARGV) {
        m/^$/      && do { next ARG };
        m/^-$/     && do { last ARG };
        s/^d//     && do { $Debug_Level++; redo OPT };
        s/^l//     && do { $Generate_Listing++; redo OPT };
        s/^i(.*)// && do { $In_Place = $1 || ".bak"; next ARG };
        say_usage("Unknown option: $_");
    }
}

```

关于循环控制操作符还有一点需要说明。你可能已经注意到，我们没有称之为“语句”。这是因为它们本来就不是语句（尽管与其他表达式类似，它们确实可以用作语句）。几乎可以认为它们是能改变控制流的一元操作符，可以在表达式中任何合理的地方使用这些循环控制操作符。实际上，甚至在不合理的地方也可以使用。有时会看到以下代码错误：

```

open FILE, '<', $file
or warn "Can't open $file: $!\n", next FILE; # 不正确

```

本意是好的，不过next FILE会解析为warn的一个参数（warn是一个列表操作符）。所以，在warn有机会发出警告之前，会先执行next。在这种情况下，这个问题很容易修正，只需要利用一些适当的括号把warn列表操作符转换为一个warn函数调用：

```

open FILE, '<', $file
or warn("Can't open $file: $!\n"), next FILE; # 正确

```

不过，你可能会发现下面的代码更可读：

```

unless (open FILE, '<', $file) {
    warn "Can't open $file: $!\n";
    next FILE;
}

```

## 裸块作为循环

BLOCK本身（有标签或无标签）在语义上等价于一个执行了一次的循环。因此，可以使

注10：可以参阅《Mastering Perl》，其中对主要的命令行参数解析模块做了比较。

用last离开这个块，或者使用redo重新开始这个块<sup>注11</sup>。需要指出，对于eval {}、sub {}或者（可能出乎很多人的意料）do {}中的块，这一点并不成立。这3个不是循环块，因为它们本身不是BLOCK；前面的关键字会让它们成为表达式中的一项，只不过其中恰好包含一个代码块而已。由于它们不是循环块，所以不能为它们指定标签，相应地也就无法应用循环控制。循环控制只能用在真正的循环上，就像return只能用在子例程（嗯，还有eval）中一样。

循环控制在if或unless中也不起作用，因为它们不是循环。不过可以多加一组大括号来提供一个裸块，这就可以算作一个循环了：

```
if (/pattern/) {{
    last if /alpha/;
    last if /beta/;
    last if /gamma/;
    # 只有当仍在f()时将在这里做一些处理
}}
```

可以采用以下方法使用代码块，使循环控制操作符可以用于一个do {}构造。为了对一个do使用next或redo，要在其中放一个裸块：

```
do {{
    next if $x == $y;
    # 在这里做一些处理
}} until $x++ > $z;
```

对于last，还要更为精细：

```
{
    do {
        last if $x = $y ** 2;
        # 在这里做一些处理
    } while $x++ <= $z;
}
```

如果希望同时使用两个循环控制，必须在这些块上加标签才能把它们区分开：

```
DO_LAST: {
    do {
DO_NEXT:      {
        next DO_NEXT if $x == $y;
        last DO_LAST if $x = $y ** 2;
        # 在这里做一些处理
    }
    } while $x++ <= $z;
}
```

不过，当然此时（或者在此之前）最好使用一个正常的无限循环（最后有一个last）：

---

注11：出于一些原因（通过反射机制可以了解这些原因，也可能无法了解），next也会退出执行一次的代码块。不过，这里稍有区别：next会执行continue块，而last不会。

```

for (;;) {
    next if $x == $y;
    last if $x = $y ** 2;
    # 在这里做一些处理
    last unless $x++ <= $z;
}

```

## 循环主题工具

Perl有很多主题工具。除了given外，还可以使用foreach循环作为主题工具。例如，可以用下面的方法统计某个字符串在一个数组中出现的次数：

```

use v5.10.1;
my $count = 0;
for (@array) {
    when ("FNORD") { ++$count }
}
print "\@array contains $count copies of 'FNORD'\n";

```

或者，更新的版本是：

```

use v5.14;
my $count = 0;
for (@array) {
    ++$count when "FNORD";
}
print "\@array contains $count copies of 'FNORD'\n";

```

在一个foreach循环中，所有when块的最后都有一个隐含的break，因为在循环中，所以这个break等价于一个next。如果你只对第一个匹配感兴趣，可以改用一个显式的last。

只有当主题在\$\_中时when才有效，所以不能指定循环变量，或者如果要指定循环变量，它必须是\$\_：

```

for my $_ (@answers) {
    say "Life, the Universe, and Everything!" when 42;
}

```

## goto操作符

不是要吓唬你（也不是要安慰你），不过Perl确实支持goto操作符。它有3种形式：goto LABEL、goto EXPR和goto &NAME。

goto LABEL形式会找到有LABEL标签的语句，并从那里继续执行。不能用这种goto跳到需要初始化的构造内部，如一个子例程或foreach循环。另外也不能跳到已经得到优化的构造中（参见第16章）。除此以外，几乎可以用它跳入当前块或动态作用域中调用goto的代码块中的任何其他地方。甚至可以跳出子例程，不过要跳出子例程，通常最好使用另外某个构



造。Perl的作者从不认为需要在Perl中使用这种需要指定标签的goto形式（除了测试这种做法确实可行）。

goto EXPR是goto LABEL的一种推广形式。它需要能生成一个标签名的表达式，标签名的位置显然要由解释器动态解析生成。因此可以像Fortran中一样“计算”goto，不过如果要提高可维护性，建议不要这样做：

```
goto(("FOO", "BAR", "GLARCH")[$i]);          # 希望 0 <= i < 3

@loop_label = qw/FOO BAR GLARCH/;
goto $loop_label[rand @loop_label];          # 随机远程端口
```

在类似这样的情况下，几乎都可以采用一种更好的想法，即利用next、last或redo的结构化控制流机制，而不是求助于goto。对于某些应用，也可以为函数提供一个引用散列，或者使用eval和die的catch/throw对完成异常处理，这些也是不错的方法。

goto &NAME形式很有魔力，从普通的goto中脱颖而出，尽管使用goto的人总是饱受诟病，但使用goto &NAME的人却能免遭非议。它用一个命名子例程调用来替换当前运行的子例程。AUTOLOAD子例程就利用这个行为来加载另一个子例程，然后假装原先调用的就是这个子例程。在goto后面，甚至caller都不能区分最早调用的是这个例程。autouse、AutoLoader和SelfLoader模块在第一次调用时都使用这个策略定义函数，然后直接跳至这些函数，别人甚至不知道这些函数根本不在那里。这不算特别轻量级，所以不要认为它是一种尾递归优化。

## 远古的Perl Case结构

在最初的20年，Perl没有正式的switch或case语句。在v5.10版本中引入given之前，人们会使用一个裸块或一次性foreach循环来建立自己的case结构。下面是一个例子：

```
SWITCH: {
    if (/^abc/) { $abc = 1; last SWITCH }
    if (/^def/) { $def = 1; last SWITCH }
    if (/^xyz/) { $xyz = 1; last SWITCH }
    $nothing = 1;
}
```

下面再给出一个例子：

```
SWITCH: {
    /^abc/ && do { $abc = 1; last SWITCH };
    /^def/ && do { $def = 1; last SWITCH };
    /^xyz/ && do { $xyz = 1; last SWITCH };
    $nothing = 1;
}
```

或者甚至可以写为：

```

if (/^abc/) { $abc = 1 }
elsif (/^def/) { $def = 1 }
elsif (/^xyz/) { $xyz = 1 }
else { $nothing = 1 }

```

在下面这个例子中，注意last操作符会忽略do{}块（这不是循环），而退出主循环：

```

for ($very_nasty_long_name[$i++][$j++]->method()) {
    /this pattern/ and do { push @flags, "-e"; last };
    /that one/ and do { push @flags, "-h"; last };
    /something else/ and do { last };
    die "unknown value: '$_'";
}

```

在比较老的Perl代码中你会经常见到这种用法，因为在given出现之前，for是编写高级主题工具的唯一方法。

不论使用哪一个主题工具，在反复比较时只指定一次值，这样不仅输入更容易，相应地也不太容易出现输入错误。这样做还可以避免再次计算表达式时可能的副作用。

?:操作符的层叠用法也可以用于简单分支（case）。下面我们再利用for的别名特性，让重复的比较更可读：

```

for ($user_color_preference) {
    $value = /red/ ? 0xFF0000 :
              /green/ ? 0x00FF00 :
              /blue/ ? 0x0000FF :
              0x000000 ; # 如果都失败则为黑色
}

```

不过，对于很多情况，最好自己建立一个散列，并在这个散列中快速索引找出答案。与我们刚才看到的层叠条件不同，散列可以扩展为有无限多个项，而且查找第一项和查找最后一项的时间相差无几。还可以在运行时增加分支。缺点是只能完成精确查找，而不能完成模式匹配。如果有一个如下的散列：

```

%color_map = (
    azure => 0xF0FFFF,
    chartreuse => 0x7FFF00,
    lavender => 0xE6E6FA,
    magenta => 0xFF00FF,
    turquoise => 0x40E0D0,
);

```

精确的字符串查找运行很快，还可以提供一个默认值：

```

$value = $color_map{ lc $user_color_preference } || 0x000000;

```

甚至多路分支语句（每个分支涉及复杂的代码块）也可以转换为快速散列查找。只需要使用由函数引用构成的一个散列，这是Perl中的首类数据类型。参见第9章中“函数散列”一节，了解如何处理这些函数散列。

前面已经说过，case结构可能并不总是工具箱中最棒的工具。要查找多态行为，最可扩展的方法是使用正常的对象分派。如果你急于了解这个内容，可以现在看看第12章。

你可能还会看到另外一种奇怪的case结构：

```
goto $data;
ABC: $foo++; goto end;
DEF: $bar++; goto end;
XYZ: $baz++; goto end;
end:
```

没错，这样是可以的，不过，嗯…这有点…慢，如果没有匹配的标签，它会查找整个程序来寻找这个没有的标签，然后结束（如果你幸运的话）。还有更好的办法结束这个语句，下一节就会介绍这样一种方法。

## 省略语句

从v5.12开始，Perl可以接受一个裸省略号“...”作为一个桩，也就是一个占位符，用来表示你还没有实现的代码。不要把这个...语句与二元触发器...操作符相混淆。Perl一般不会把它们搞错，因为大多数情况下Perl都能分清什么时候需要语句，什么时候想要操作符，请看下面的说明。

Perl解析省略语句时，它会悄悄地接受这个语句。不过，以后如果你想要执行这个语句，Perl会“高调地”抛出一个异常，并提供文本Unimplemented，表示这个代码未实现：

```
sub unimplemented { ... }
eval { unimplemented() };
if ($@ eq "Unimplemented") {
    say "Caught an Unimplemented exception!";
}
```

可以把省略语句当作一个完整的语句（不过允许有一个语句修饰符）。下面这些例子都是省略语句的合法例子：

```
{ ... }
sub foo { ... }
...;
eval { ... };
... unless defined &dispatcher;
sub somemeth {
    my $self = shift;
    ...;
}
$x = do {
    my $n;
    ...;
    say "Hurrah!";
    $n;
}
```



};

不过，...不能表示作为一个更大语句一部分的表达式，因为...（三个点）同时也是一个触发器操作符，见第3章中“范围操作符”一节的介绍。因此，以下代码都认为存在语法错误：

```
print ...;                # 不正确
open(my $fh, ">", "/dev/passwd") or ...;    # 不正确
if ($condition && ... ) { say "Howdy" };    # 不正确
```

有时Perl不能区分表达式还是语句。例如，一个裸块和一个匿名散列构造器看起来就是一样的，除非大括号里有些其他内容，可以为Perl提供必要的提示：

```
@transformed = map { ... } @input;    # 不正确：语法错误
```

一种解决方法是在块中使用一个“;”，告诉Perl这个{ ... }是一个块，而不是一个匿名散列构造器：

```
@transformed = map {; ... } @input;    # ; 消除省略号歧义
@transformed = map { ...; } @input;    # ; 消除省略号歧义
```

人们通俗地把这个标点符号称为“yada-yada”（表示“等等，诸如此类”），不过如果想表达这种感觉，可以直接用它的技术名“省略号”。Perl还不接受Unicode名U+2026 HORIZONTAL ELLIPSIS作为...的别名，不过可能将来某一天就会接受…

## 全局声明

子例程和格式声明都是全局声明。不论把这些声明放在哪里，所声明的东西都将是全局的（它们对包来说是局部的，不过包对于程序而言则是全局的，所以包中的所有内容都将全局可见）。能放语句的地方都可以放全局声明，不过它对主要语句序列的执行没有任何影响，声明只在编译时起作用。

这说明，不能有条件地声明子例程或格式，也就是说，要想把子例程或格式隐藏在一个运行时条件语句中（如if），希望编译器看不到，这是办不到的，因为只有解释器会注意到这些条件。不论子例程和格式声明（以及use和no声明）出现在哪里，编译器都会看到。

全局声明通常放在程序的开始或末尾位置，或者放在其他文件中。不过，如果要声明词法作用域变量（见下一节），需要确保格式或子例程定义在这些变量声明的作用域内（如果希望它能访问那些私有变量）。

需要注意，我们悄悄地将话题从声明转到了定义。有时将子例程的定义与声明分开会有很大帮助。二者唯一的语法区别是：定义提供了一个BLOCK，其中包含要执行的代码，而声明没有（如果没有看到声明，则子例程定义相当于自己的声明）。通过将定义与声明划分

开，这样可以把子例程声明放在文件最前面，而把定义放在最后面（词法作用域变量声明可以放在中间）：

```
sub count (@);      # 编译器现在知道如何调用count()
my $x;              # 编译器现在知道了词法作用域变量
$x = count(3,2,1);  # 编译器可以验证函数调用
sub count (@) { @_ } # 编译器现在知道count()是什么意思
```

如上面这个例子所示，编译子例程调用之前并不一定先定义这些子例程（实际上，如果使用自动加载，甚至可以延迟到第一次使用子例程时再提供定义），不过声明子例程会在很多方面对编译器很有帮助，另外对于如何调用子例程可以提供更多选项。

如果声明了一个子例程，编译时在这个声明后面使用子例程时可以不加括号，就好像它是一个内置的操作符（上一个例子中我们调用count时使用了括号，不过实际上并不需要这么做）。可以声明一个子例程而不提供定义，如下所示：

```
sub myname;
$me = myname $0 or die "can't get myname";
```

类似这样的裸声明会声明这个函数是一个列表操作符，而不是一个一元操作符，所以注意这里要使用or而不是||。||操作符在列表操作符后面使用时绑定过于紧密，不过可以使用括号将列表操作符的参数括起来，从而将这个列表操作符再转换为一个函数调用。或者，可以使用原型(\$)将子例程转换为一个一元操作符：

```
sub myname ($);
$me = myname $0 || die "can't get myname";
```

现在就会如你期望的那样解析，不过还是要养成在这种情况下使用括号的习惯。有关原型的更多内容，请参见第7章。

还是需要在某个地方定义子例程，否则会在运行时得到一个错误，指示你调用了一个未定义的子例程。除了自己定义子例程外，还有很多方法可以从别处得到子例程定义。

可以利用一个简单的require语句从其他文件加载定义；这是Perl 4中加载文件的最佳方法，不过这种方法存在两个问题。首先，那个文件通常会把子例程名插入到它自己选择的一个包（符号表）中，而不是插入到你的包中。其次，require会运行时执行，所以它发生得太晚，不能作为调用require的文件中的一个声明。不过，有时延迟加载正是你想要的。

要引入声明和定义，一种更有用的方法是利用use声明，实际上这会在编译时用require加载模块（因为use相当于一个BEGIN块），然后将一些模块声明导入到你自己的程序中。因此，可以把use认为是一种全局声明，因为它可以在编译时将名字导入到你自己的（全局）包中，就好像这些是你自己声明的一样。参见第10章的“符号表”一节，来了解在不同的包之间实现导入的底层机制，另外可以参见第11章了解如何建立模块的导入和导



出机制，还可以参见第16章，其中对BEGIN和相应的CHECK、UNITCHECK、INIT和END做了解释，在某种程度上这些也是全局声明，因为它们都在编译时处理，会有全局影响。

## 作用域声明

类似于全局声明，词法作用域声明在编译时起作用。与全局声明不同的是，词法作用域声明只应用于从声明到最内层外围作用域末尾（可能是块、文件或eval，要看哪一个最先出现）的范围内。正是因为这个原因，我们把它们称为词法作用域声明，不过称为“文本作用域”可能更准确，因为词法作用域与词法分析没有什么关系。不过全世界的计算机科学家都知道“词法作用域”是什么意思，所以这里也沿用了这种说法。

Perl还支持动态作用域声明。动态作用域（dynamic scope）也延伸到最内层外围块的末尾，不过在这种情况下，“外围”是在运行时动态定义的，而不是在编译时指定。换种说法来讲，块要通过调用其他块动态嵌套，而不是通过包含其他块来实现嵌套。这种动态作用域嵌套可能与词法作用域嵌套有些关联，不过二者通常并不相同，特别是在调用子例程时。

我们提到过，use的某些方面可以认为是全局声明，不过use的另外一些方面却是词法作用域声明。具体地，use不只是导入包符号，它还会实现各种有魔法的编译器提示，这也称为*pragmas*（如果你懂希腊语，也就是*pragmata*）。大多数*pragma*都是词法作用域声明，包括我们不时提到的strict *pragma*。参见后面的“Pragmas”一节（因此，如果在文件最前面使用use v5.14隐式打开strict *pragma*，则它会作用于其余的整个文件，即使切换了不同的包）。

有意思的是，尽管包（package）是一个全局实体，但包声明本身是词法作用域声明。不过package声明只为外围块的其余部分声明这个缺省包的身份，或者，如果在包NAMESPACE后面使用了可选的BLOCK，则是在这个特定的块中声明这个缺省包。变量名<sup>注12</sup>中使用的未声明标识符都会在这个包中查找。从某种意义上讲，实际上根本不会声明包，而是提到属于一个包的某个东西时，这个包就会出现。这就是Perl风格。

## 作用域变量声明

本章后面主要讨论如何使用全局变量。或者，更确切地讲，我们要介绍如何不使用全局变量。有很多声明可以帮助你不使用全局变量，或者至少不要愚蠢地使用。

我们已经提到过package声明，这在很早以前就引入了Perl，允许把全局变量分开放在不同的包中。这对于某些类型的变量很适合。库、模块和类使用包来存储它们的接口数据（以及它们的一些半私有数据），以避免与主程序或其他模块中同名的变量和函数冲突。如果

---

注12：以及子例程、文件句柄、目录句柄和格式的未限定名。



看到有人写了`$Some::stuff`<sup>注13</sup>，说明他在使用包`Some`中的`$stuff`标量变量。请参见第10章。

如果完全这么做，随着Perl程序越来越长，很快会变得很笨拙。幸运的是，利用Perl的三种作用域声明符，可以很容易地创建完全私有的变量（使用`my`或`state`），或者允许有选择地访问全局变量（使用`our`）。另外还有一个伪声明符，可以为全局变量提供临时值（使用`local`）。这些声明符要放在相应的变量前面：

```
my $nose;
our $House;
state $troopers = 0;
local $TV_channel;
```

如果要声明多个变量，变量列表必须放在括号里：

```
my ($nose, @eyes, %teeth);
our ($House, @Autos, %Kids);
state ($name, $rank, $serno);
local (*Spouse, $phone{HOME});
```

`my`、`state`和`our`声明符只能声明简单的标量、数组或散列变量，而`state`只能初始化简单的标量变量（不过这个变量可以包含另外某个对象的引用），而不能初始化数组或散列。由于`local`不是一个真正的声明符，这些约束稍稍宽松一些：它能用来局部化整个类型团和单个元素，或者局部化数组和散列的片段，可以在局部化的同时初始化（也可以不初始化）。这些修饰符分别为它们修饰的变量提供了不同类型的“限制”。简单地说：`our`将名字限制于某个作用域，`local`将值限定于某个作用域，而`my`将名字和值都限制于某个作用域（`state`类似于`my`，不过它定义作用域的方式稍有不同）。这些构造都可以赋值，不过在值的具体处理方面有所区别，因为它们会采用不同的机制存储值。如果没有赋任何值（就像上面一样），它们也有区别：`my`和`local`会适当地将所涉及的变量赋一个初始值`undef`或`()`；与之不同，`our`会保留相关全局变量的当前值不变。最奇怪的是`state`，它会以上一次在该位置存储的值作为初始值。

从语法来讲，`my`、`our`、`state`和`local`都是左值的修饰符（就像形容词一样）。为一个有声明符修饰的左值赋值时，不会改变这个左值的类型（即看作一个标量还是一个列表）。要确定如何赋值，只要假装声明符不存在。所以：

```
my ($foo) = <STDIN>;
my @array = <STDIN>;
```

都会为右边提供列表上下文，而以下代码会提供标量上下文：

```
my $foo = <STDIN>;
```

---

注13：或者很早以前的写法`$Some'stuff`，除了Perl，可能不建议这种写法。

声明符比逗号绑定更紧密（有更高的优先级）。下面的例子有错误，它只声明了一个变量，而不是两个，因为声明符后面的列表没有用括号括起来：

```
my $foo, $bar = 1; # 不正确
```

这与以下代码有同样的效果：

```
my $foo;
$bar = 1;
```

如果打开了`strict`，你会得到一个错误，因为`$bar`未声明。

一般来讲，最好在适合的最小作用域中声明变量。由于控制流语句中声明的变量只在该语句控制的块中可见，所以它们的可见性范围会缩减。在英语中这样也更易读：

```
sub check_warehouse {
    for my $widget (our @Current_Inventory) {
        say "I have a $widget in stock today.";
    }
}
```

到目前为止，最常看到的声明符是`my`，它会声明词法作用域变量，这些变量的名字和值都存储在当前作用域的临时便签簿中，不能从词法作用域之外访问这些变量。要尽量使用`my`，除非你很清楚出于什么理由确实有必要使用另外某个声明符。如果你想有同样的私有性，但是还希望能够在多个调用之间保留变量的值，则可以使用`state`。

与之紧密相关的是`our`声明，这也是持久的，会在当前作用域中输入一个词法作用域名，不过其持久性是通过将值存储在一个全局变量中来实现的，这样任何人只要愿意都可以访问这个变量。换句话说，这是一个全局变量，只不过伪装成一个词法作用域变量。

除了全局作用域和词法作用域，我们还有一种动态作用域（dynamic scoping），这由`local`实现，不过“`local`”实际上处理的是全局变量，与局部便签簿没有任何关系（如果命名为`temp`会更合适，因为它会临时改变一个现有变量的值。Perl 6中就有这个关键字，如果从Perl 6借用，没准有一天你会在Perl 5程序中看到`temp`）。

新声明的变量[或值（对于`local`来说）]并不会立即出现，而是在相应声明语句后面的语句中才会出现。因此，可以采用以下方式镜像一个变量：

```
my $x = $x;
```

这会将新的内部变量`$x`初始化为当前值`$x`，而不论`$x`目前是全局变量还是词法作用域变量。

声明一个词法变量名会隐藏之前声明的所有同名词法作用域变量，不论那些变量是在这个作用域中还是在外部作用域中声明（不过，如果启用了警告，这会使你得到一个警告）。它还会隐藏所有同名的未限定全局变量，不过通过显式限定全局变量，即加上全局变量所在包的包名，总能访问到这个全局变量，例如`$PackageName::varname`。



## 词法作用域变量：my

为了帮你摆脱维护全局变量的痛苦，Perl提供了词法作用域变量，通常简写为lexicals。与全局变量不同，词法作用域变量可以保证私有性。假设你没有把这些私有变量的引用传出去（允许别人间接地修改这些变量），就能确保每一次访问这些私有变量都限制在程序中一个很容易标识的独立代码中。毕竟，这也是我们选择关键字my的原因。

语句序列可以包含词法作用域变量的声明。这些声明往往放在语句序列最前面，不过这不是一个严格的要求；你也可以在变量第一次使用时用一个my声明符来装饰（只要在使用这个变量的最外层作用域中）。除了在编译时声明变量名以外，这些就像是正常的运行时语句：它们会在语句序列中逐个执行，就好像是没有声明符的正常语句：

```
my $name = "fred";
my @stuff = ("car", "house", "club");
my ($vehicle, $home, $tool) = @stuff;
```

这些词法作用域变量对其直接外围作用域以外的世界完全是隐藏的。与local的动态作用域效果不同（见下面的介绍），词法作用域变量对从其作用域调用的所有子例程都是隐藏的。即使从自身或其他地方调用同一个子例程也是如此——子例程的每一个实例都有其自己的词法作用域变量“便签簿”。不过，就像任何内部作用域一样，在一个词法作用域变量所在的作用域中定义的子例程可以看到这个变量。

与块作用域不同，文件作用域不嵌套；这里没有“外围”的概念，至少从词法作用域的角度来讲不存在“外围”的概念。如果用do、require或use从一个单独的文件加载代码，这个文件中的代码不能访问你的词法作用域变量，你也不能访问那个文件中的词法作用域变量。

不过，一个文件中的任何作用域（甚至这个文件自身）都是平等的。如果子例程的作用域比子例程定义更大，这通常很有用，因为这样就能在有限的一组子例程之间共享一些私有变量。要创建C程序员所认为的“文件静态”变量，就可以采用这种方法：

```
{
    my $state = 0;

    sub on { $state = 1 }
    sub off { $state = 0 }
    sub toggle { $state = !$state }
}
```

eval STRING操作符也可以作为嵌套作用域，因为eval中的代码可以看到其调用者的词法作用域变量（只要这些名字没有被eval自己作用域中同名的声明所隐藏）。匿名子例程可以类似地访问其外围作用域的词法作用域变量；如果是这样，其外围作用域也称为闭包



(closures)<sup>注14</sup>。结合这两个概念，如果一个块使用eval执行一个字符串（它会创建一个匿名子例程），那么这个子例程就会成为一个闭包，允许访问eval和这个块的词法作用域变量，甚至在退出eval和块之后仍能访问。有关内容参见第8章中“闭包”一节。

## 持久词法作用域变量：state

state变量也是词法作用域变量，这与my变量类似。唯一的区别是，state变量不会重新初始化，而my变量有所不同，每次进入my变量的外围块时都会重新初始化my变量。由于state变量有这个特性，所以才能保证函数可以有一个“持久的”私有变量，在多次调用这个函数之间，这个变量仍保留原来的值。

只有当use feature "state" pragma起作用的情况下才会启用state变量。如果用use声明要求使用Perl v5.10及以后的版本，会自动包含这个pragma：

```
use v5.14;
sub next_count {
    state $counter = 0; # 仅第一次通过
    return ++$counter;
}
```

与my变量不同，state变量目前仅限于标量变量；它们不能是数组或散列。听上去好像有很大局限性，不过实际上并非如此，因为完全可以在一个state变量中存储数组或散列的引用：

```
use v5.14;
state $bag = { };
state $vector = [ ];

...
unless ($bag->{$item}) { $bag->{$item} = 1 }
...
push @$vector, $item;
```

## 词法作用域全局声明：our

在use strict出现之前的日子里，Perl程序可以直接访问全局变量。如今，访问全局变量更好的方法是利用our声明。这个声明有词法作用域，因为它只应用到当前作用域的末尾。不过，与词法作用域my或动态作用域local不同，our不会将任何东西隔离到当前词法或动态作用域。实际上，它会在当前词法作用域中提供“权限”，允许访问当前包中指定一个变量。由于它声明了一个词法作用域变量名，所以会隐藏之前所有同名的词法作用域变量。在这个方面，our变量就相当于my变量。

---

注14：为了便于记忆，请注意“外围作用域”和“闭包”之间的公共元素（这两个词中都有表示关闭的“clo”）。闭包的具体定义来自一个数学概念，主要考虑值集合及对这些值的操作的完备性。

如果把一个our声明放在用大括号限定的块之外，它会一直持续到当前编译单元的末尾。不过，通常人们会把它放在一个子例程定义的最上面，以表明在访问一个全局变量：

```
sub check_warehouse {
    our @Current_Inventory;
    my $widget;
    foreach $widget (@Current_Inventory) {
        say "I have a $widget in stock today.";
    }
}
```

由于全局变量比私有变量生存期更长，而且可见范围更大，所以不同于临时变量，我们喜欢为全局变量使用更长而且更有意义的名字。如果认真遵循这个实践做法，几乎能像use strict一样防止过度使用全局变量，特别是那些愿意兢兢业业打字的程序员。

重复的our声明并不表示嵌套。每个嵌套的my都会生成一个新变量，而每一个嵌套的local会生成一个新值。不过，每次使用our时，还是指示同一个全局变量，而不论是否嵌套。为一个our变量赋值时，这个赋值的效果会持续到声明作用域之后。这是因为，our不会创建值。它只是为全局变量提供了一种受限的访问，而全局变量本身会一直存在：

```
our $PROGRAM_NAME = "waiter";
{
    our $PROGRAM_NAME = "server";
    # 这里调用的代码会看到"server"
    ...
}
# 这里执行的代码仍看到"server"
```

可以与my或local声明做个比较，对于my或local声明，在块之后，外部变量或值将再次可见：

```
my $i = 10;
{
    my $i = 99;
    ...
}
# 这里编译的代码会看到外部变量值10

local $PROGRAM_NAME = "waiter"
{
    local $PROGRAM_NAME = "server"
    # 这里调用的代码会看到"server"
    ...
}
# 这里执行的代码看到恢复的"waiter"值
```

对于our声明的变量，通常只赋值一次，可能是在程序或模块的最前面，或者也可以在our前面加上一个local（这种情况比较少见）：

```
{
```

```

    local our @Current_Inventory = qw(bananas);
    check_warehouse(); # 不，我们没有香蕉
}

```

不过在这种情况下，为什么不直接作为一个参数传递呢？

## 动态作用域变量：local

对一个全局变量使用local操作符时，会在每次执行local时为它提供一个临时值，不过这不会影响这个变量的全局可见性。程序到达这个动态作用域末尾时，这个临时值就会被丢弃，而恢复为原来的值。不过这仍然是一个全局变量，只不过在执行这个块时有一个临时值。如果你在全局变量有一个临时值时调用另外某个函数，而且这个函数会访问这个全局变量，它将看到这个临时值，而不是原来的值。换句话说，这个函数会在你的动态作用域中，尽管它原本不在你的词法作用域中<sup>注15</sup>。

这个过程称为建立动态作用域（dynamic scoping），因为全局变量的当前值取决于你的动态上下文；也就是说，它取决于调用链中哪一个父调用可能会调用local。调用前的最后一个上下文将控制所看到的值。

如果有一个local变量，如下所示：

```

{
    local $var = $newvalue;
    some_func();
    ...
}

```

可以单纯地按运行时赋值来考虑：

```

{
    $oldvalue = $var;
    $var = $newvalue;
    some_func();
    ...
}
continue {
    $var = $oldvalue;
}

```

使用local的区别在于，不论如何退出块（甚至永久地从这个作用域返回），都会恢复原来的值。

与my类似，可以将一个local变量初始化为相同全局变量的一个副本。子例程（以及在这

---

注15：正是因为这一点，有时词法作用域也称为静态作用域（static scopes），以区别动态作用域，并强调它是在编译时确定的。不要把这个术语与C或C++中static的用法混淆了。这个词有很多不同的含义，这也是我们避免使用这个术语的原因。



个子例程中调用的任何其他子例程，它们当然也可以看到这个动态作用域全局变量）执行期间对该变量的任何修改都会被子例程返回时被丢弃。不过，最好对你要做的工作加个注释：

```
# 警告：对这个动态作用域的改变是临时的
local $Some_Global = $Some_Global;
```

不论是用`our`显式声明，还是允许其自然存在，另外尽管它存放的局部值肯定会在作用域退出时丢掉，不管怎样，全局变量仍在整个程序中可见。在小程序中这还不算是问题，不过对于大程序，你很快就会忘记到底在代码中哪些地方用到了这些全局变量。如果愿意，可以通过`use strict 'vars' pragma`（见下一节的介绍）禁止随意地使用全局变量。

尽管`my`和`local`都会提供某种程度的保护，总的说来，我们更倾向于`my`而不是`local`。不过，有时必须使用`local`，以便临时地改变一个现有全局变量的值，如第25章所述。因为只有字母数字标识符可以是词法作用域变量，而很多特殊变量并不一定是字母数字标识符。你还需要使用`local`对一个包的符号表做些临时修改，参见第10章“符号表”一节介绍。最后一点，可以在一个数组或散列的单个元素或整个片段上使用`local`。甚至这个数组或散列恰好是一个词法作用域变量时也可以这么做，这样会在这些词法作用域变量之上增加`local`的动态作用域行为。我们不打算这里太多地讨论`local`的语义。有关的更多信息请见第27章中对`local`的介绍。

## Pragmas

很多编程语言允许你为编译器提供一些提示。在Perl中，这些提示要用`use`声明提供给编译器。以下给出了一些`pragma`：

```
use warnings;
use strict;
use integer;
use bytes;
use constant pi => ( 4 * atan2(1,1) );
```

Perl `pragma`将在第29章详细介绍，不过现在重点介绍在本章讨论的内容中最有用的两个`pragma`。

尽管很多`pragma`是全局声明，会影响全局变量或当前包，但大多数是词法作用域声明，其效果只持续到外围块、文件或`eval`（要看哪一个最先出现）的末尾。词法作用域`pragma`可以在一个内部作用域中用`no`声明取消，`no`声明与`use`类似，但作用相反。

## 控制警告

为了显示这是如何工作的，我们将使用`warnings pragma`告诉Perl是否对有问题的做法发出警告：

```

use warnings;          # 显式启用warnings
...
{
    no warnings;       # 禁用warnings, 直到块结束
    ...
}
# 这里再次自动启用warnings

```

一旦启用warnings, Perl会对有问题的情况发出警告, 如变量只使用了一次, 变量声明屏蔽了相同作用域中的其他声明, 字符串不适当地转换为数字, 使用未定义的值作为合法字符串或数字, 试图写一个按只读方式打开 (或者根本没有打开) 的文件, 以及perldiag中描述的很多其他情况。

warnings pragma是控制警告的首选方法。较早的程序只能使用-w命令行开关, 或者修改全局\$^W变量:

```

{
    local $^W = 0;
    ...
}

```

使用use warnings和no warnings pragma会更好。pragma更好是因为它在编译时发生, 由于这是一个词法作用域声明, 因此不会影响它不想影响的代码, 另外还支持对各类警告进行细粒度的控制 (不过我们未在这些简单的例子中展示这一点)。关于warnings pragma的更多信息, 包括如何将一般警告转变为致命错误, 以及如何覆盖这个pragma在全局范围打开警告 (甚至包括拒绝打开警告的模块), 请参见第29章的“警告”一节。

## 控制全局变量的使用

另一个常见的声明是strict pragma, 它有很多功能, 其中之一就是控制全局变量的使用。正常情况下, 只要你提到新的全局变量, Perl就会创建这些全局变量 (或者, 更常见的, 会覆盖旧的全局变量)。默认情况下没有必要提供变量声明。由于这种不加约束地使用全局变量可能会使很大的程序或模块维护起来很痛苦, 有时你可能希望不允许这种随意地使用全局变量。为了帮助防止这种意外发生, 可以声明:

```

use v5.14;          # 隐式地打开strict
use strict "vars";  # 显式打开strict

```

这意味着, 从这里到外围作用域末尾提到的所有变量必须是一个用my、state或our声明的词法作用域变量, 要么是一个已明确允许的全局变量。如果不属于这两种情况, 就会得到一个编译错误。如果以下条件满足则认为全局变量是明确允许的:

- 是Perl的程序级特殊变量之一 (参见第25章)。
- 用其包名完全限定 (参见第10章)。

- 已经导入当前包（参见第11章）。
- 通过一个`our`声明伪装为一个词法作用域变量（这是我们在Perl中增加`our`声明的主要原因）。

当然，肯定还有第5种可能性，如果觉得这个`pragma`太麻烦，只需要在一个内部块中使用以下声明将其取消：

```
no strict "vars";
```

还可以用这个`pragma`打开检查，即对解除符号引用和随意使用裸字的情况要进行严格检查。正常情况下，只需要声明：

```
use strict;
```

就可以启用所有这3个结构（如果还没有通过`use v5.14`或类似声明隐式地启用）。有关的信息请参见第29章对“`strict`” `pragma`的介绍。



# 模式匹配

利用Perl提供的内置模式匹配支持，你可以方便而高效地搜索大量数据。不论是大型商业门户网站希望扫描每一条信息来找出人们感兴趣的新闻报道；还是政府机构致力于研究人口统计学（或人类基因组）；或是一个教育机构希望在网站上随时公布动态信息，Perl都是上选的工具，部分原因在于它的数据库连接，不过更大程度上是因为它的模式匹配功能。如果从最宽泛的意义上理解“文本”，你所做的90%的工作中，可能有90%都是文本处理。这就是Perl的核心，也是我们一直以来所强调的。实际上，这甚至是Perl名字的一部分，Perl的全名是：实用摘录与报表语言（*Practical Extraction and Report Language*）。Perl的模式为搜索海量的数据和从中摘录有用的信息提供了一个强大的工具。

可以创建一个正则表达式（regular expression，或regex）来指定模式，Perl的正则表达式引擎（本章后面将简称为“引擎”）根据这个正则表达式来确定模式与你的数据是否匹配（以及如何匹配）。尽管大部分数据都可能是文本串，不过完全可以使用正则表达式来搜索和替换字节序列，甚至可以用来匹配你通常所认为的“二进制”数据。对于Perl来说，字节就是字符，只不过它们的序数值碰巧小于256而已（有关的更多内容请参见第6章）。

如果你通过其他途径已经对正则表达式有所了解，那么我们要提醒你：Perl中的正则表达式稍有不同。首先，从这个词的理论意义上讲，正则表达式并不完全是“规则的”，这意味着，相比于计算机科学课程上所讲的传统正则表达式，Perl的正则表达式可以做更多工作。其次，正则表达式在Perl中实在太常用，因此已经紧密地集成到这个语言中，而不是像其他库那样松散地接入。刚接触Perl的程序员常常会查找类似下面的函数，最后却徒劳无功，因为根本没有这样的函数：

```
match( $string, $pattern );  
subst( $string, $pattern, $replacement );
```

由于匹配和替换在Perl中是非常基本的任务，所以它们有自己的单字母操作符：

`m/PATTERN/`和`s/PATTERN/REPLACEMENT/`（简写为`m//`和`s///`）。它们不仅在语法上更为简洁，而且可以像双引号字符串一样解析，而不是作为普通的操作符；不过，它们的操作仍与操作符类似，所以我们还是把它们叫做操作符。在这一章中，你会看到这些操作符将用来对字符串完成模式匹配。如果字符串的某一部分与模式匹配，我们就称这个匹配成功。对于一个成功的模式匹配，你可以做很多很酷的处理。具体来讲，如果在使用`s///`，成功匹配会使字符串中匹配的部分替换为`REPLACEMENT`指定的内容。

这一章主要讨论如何构造和使用模式。Perl的正则表达式非常高效，可以把很多含义压缩在一个很小的空间里。因此，如果你想凭直觉看出一个长模式的含义，这会很困难，可能会让你望而却步。不过如果能把长模式分解为多个部分，而且你知道引擎如何解释这些部分，那么理解任何正则表达式都不成问题。一个上百行代码的C或Java程序可能用一行Perl正则表达式就可以表示，这种情况并不少见。这个正则表达式理解起来可能会比长程序中的单个代码行稍难一些；但另一方面，从整体来看，与理解整个程序相比，理解这个正则表达式可能要容易得多。你要正确地看待这两方面。

## 正则表达式家族

在深入讨论解释正则表达式的规则之前，下面来看模式到底是怎样的。正则表达式中的大多数字符只与自身匹配。如果把多个字符连续串在一起，正如你希望的那样，它们必须按顺序匹配。所以，如果写出以下模式匹配：

```
/Frodo/
```

可以确信，除非字符串中某个位置包含有子串“Frodo”，否则这个模式不会匹配（子串（substring）就是字符串的一部分）。在字符串中，任何位置上都可以发生匹配，只要在这个位置上这5个字符按以上顺序先后出现。

其他字符不与自身匹配，而会以某种方式表现出一些“奇怪的行为”。我们把这些字符称为元字符（metacharacters）。所有元字符本身都很淘气，不过有些更“坏”，会把它附近的字符也带坏，让它们也有奇怪的行为。

以下就是这些“异类”：

```
\ | ( ) [ { ^ $ * + ? .
```

实际上元字符非常有用，它们在模式中有一些特殊的含义；在后面的介绍中，我们会逐步介绍所有这些含义。不过，为了让你安心，首先可以告诉你：只要在这些字符前面分别加一个反斜线，就能匹配这12个字符。例如，反斜线本身就是一个元字符，所以要匹配一个真正的反斜线，需要在反斜线前面再加一个反斜线：`\\`。

可以看到，反斜线就是那种“异类”字符，会让其他字符有奇怪的行为。实际上，如果你



让一个原本有奇怪行为的元字符表现得与平常不一样，最终它就会有正常的行为，正所谓负负得正。所以如果在一个字符前面加反斜线，会让它表现它的本来含义，不过这只适用于标点字符；如果在一个（正常的）字母数字字符前面加反斜线，则正好相反：它会让原本正常的字符变得特殊。只要看到如下的两字符序列：

```
\b \d \t \3 \s
```

你应该知道，这个序列是一个元符号（metasymbol），将匹配某个奇怪的东西。例如，`\b`匹配一个单词边界，而`\t`匹配一个正常的制表符。注意，制表符宽度为一个字符，不过单词边界字符宽度为0，因为它只是两个字符之间的一个位置。所以我们将`\b`称为一个0宽度断言。不过，`\t`和`\b`有一点是相似的，它们都是关于字符串中某个特定位置的断言。对一个正则表达式中的某个内容断言（assert）时，就是在宣布要想让这个模式匹配，这个特定的东西必须为true。

正则表达式中的各个部分大多是某种断言，包括普通字符（它们断言与自身匹配）。更确切地讲，它们还会断言下一个内容要匹配字符串中的下一个字符，正是因为这个原因，我们说制表符的宽度为“一个字符”。有些断言（如`\t`）在匹配时会“吞掉”一部分字符串，而另外一些断言（如`\b`）不会这样。不过我们通常还是将0宽度断言称为断言。为了避免混淆，我们把有宽度的断言称为原子（atom）。（如果你是一个物理学家，可能会认为非0宽度的原子有质量，相对地，0宽度断言就像光子，没有质量。）

还可以看到一些元字符不是断言，实际上，它们是结构字符（如大括号和分号会定义正常Perl代码的结构，但自己实际上不做任何事情）。在某些方面，这些结构元字符可以算是最重要的元字符，因为学习读正则表达式时，关键的第一步就是要学会用眼睛挑出这些结构元字符。一旦学会这一点，读正则表达式就是小菜一碟了<sup>注1</sup>。

竖线就是这样一个结构元字符，它指示候选项（alternation）：

```
/Frodo|Pippin|Merry|Sam/
```

这说明，这些字符串都可以触发一个匹配；这个内容将在这一章后面的“候选项”一节中介绍。另外在“分组和捕获”一节中，我们会展示如何使用小括号包围模式中的某些部分来建立分组：

```
/(Frodo|Drogo|Bilbo) Baggins/
```

或者甚至可以：

```
/(Frod|Drog|Bilb)o Baggins/
```

你还会看到另外一种东西，我们称之为量词（quantifiers），它们指示前一个东西要连续匹配多少次。量词如下所示：

---

注1： 必须承认，有时也不是那么轻松，不过也不会太困难。



`* + ? *? *+ {3} {2,5}`

不过，它们绝对不会像这样单独出现。量词只有与原子关联时才有意义。也就是说，只对有限宽度的断言适用<sup>注2</sup>。量词只能关联到前面的原子，按人类语言来讲，这表示正常情况下它们只指定前面一个字符的数量。如果你想连续匹配“bar”的3次出现，需要用小括号把“bar”的单个字符分组到一个“分子”中，如下所示：

`/(bar){3}/`

这样就能匹配“barbarbar”。如果写为`/bar{3}/`，这会匹配“barrrr”。说得好听点，会认为你是苏格兰人，说得不好听，可能会说你是野蛮人（也可能不会。我们喜欢的一些元字符就像苏格兰语）。关于量词的更多内容，请参见这一章后面的“量词”一节。

现在你已经了解了构成正则表达式的一些“野兽”，可能迫不及待地想要驯服它们。不过，在仔细讨论正则表达式之前，需要先退回去，谈一谈使用正则表达式的模式匹配操作符（如果在这个过程中你又看到更多正则表达式野兽，正好可以作为技巧留给后面的学习之旅）。

## 模式匹配操作符

从动物学角度来讲，Perl的模式匹配操作符相当于一种为正则表达式提供的笼子，可以把正则表达式关起来，不让它们出来。这是有意设计的；如果让这些正则表达式野兽在语言中到处乱跑，Perl就完全变成了一个丛林。当然，这个世界需要有自己的丛林（毕竟，丛林都是有多多样性的引擎），不过丛林应该留在自己的位置上。类似的，尽管正则表达式具有组合多样性，但它们应该留在模式匹配操作符里面，这才是它们应该去的地方。这就是它们的丛林。

正则表达式还不够强大，`m//`和`s///`操作符还提供了双引号内插能力（同样受限）。由于模式会像双引号字符串一样解析，所有双引号约定仍然适用，包括变量内插（除非你使用单引号作为定界符）和用反斜线转义的特殊字符（参见这一章后面的“特殊字符”一节）。将字符串解释为正则表达式之前会先应用这些约定（Perl语言中字符串很少需要经过多轮处理，不过这里就属于这种情况，需要对字符串做多次处理）。第一轮处理不算是常规的双引号内插，因为它知道哪些要完成内插，而哪些需要传递给正则表达式解析器。例如，如果`$`后面紧跟着一个竖线、结束小括号或字符串末尾，这就不会处理为一个变量内插，而是要处理为表示行尾的传统正则表达式断言。所以，如果有：

```
$foo = "bar";  
/$foo$/;
```

---

注2：量词有点像第4章介绍的语句修饰符，只能与单个语句关联。如果将量词关联到一个0宽度断言，这就像把一个while修饰符关联到一个声明，这些做法都没有任何意义，就好像想从当地药房买一磅光子，要知道药房只卖原子之类的东西。

双引号内插知道这两个\$号的作用不同。它会完成\$foo的内插，然后把它交给正则表达式解析器：

```
/bar$/;
```

这种两轮解析还有另一个结果，普通的Perl词法分析器首先找到正则表达式的末尾，就好像它在查找一个正常字符串的终止定界符一样。只有在找到字符串末尾（并完成所有变量内插）之后，才会把模式当作一个正则表达式。这说明，不能把一个模式的终止定界符“隐藏”在正则表达式构造中（如一个中括号字符类或一个正则表达式注释，这些我们还没有介绍）。Perl会看到这个定界符，并在这个位置终止模式。

还要知道，如果变量的值会不断变化，将这种变量内插到一个模式时，这会减慢模式匹配工具的速度（它有可能必须重新编译模式）。参见这一章后面的“变量内插”一节。可以用很早以前的/o修饰符硬性禁止再编译，不过通常更好的办法是使用qr//构造抽出这些改变的部分，从而只重新编译那些真正需要再编译的部分。

tr///转换操作符不会内插变量，它甚至不使用正则表达式（实际上，转换操作符可能根本不应该放在这一章中介绍，不过我们实在想不出更好的地方来介绍它）。不过，它与m//和s///确实有一个共同的特性，它也使用=~和!~操作符绑定到变量。

=~和!~操作符已经在第3章中介绍过，可以把左边的标量表达式绑定到右边的3个类引号操作符之一：m//用于匹配一个模式，s///用某个字符串替换与一个模式匹配的子串，tr///（或其同义词y///）用于把一个字符集转换为另一个字符集（如果斜线用作定界符，也可以把m//写为//，不加m）。如果=~或!~的右边不是这3个操作符，仍会当作一个m//匹配操作，不过这样一来，将没有地方放置任何末尾修饰符（见下一节“模式修饰符”），而且你还必须处理你自己的引号机制：

```
say "matches" if $somestring =~ $somepattern;
```

实际上，实在没理由不明确指定这个操作符：

```
say "matches" if $somestring =~ m/$somepattern/;
```

用于匹配操作时，=~和!~有时分别读作“匹配”和“不匹配”（不过如果读作“包含”和“不包含”可能含义更清楚）。

除了m//和s///操作符，正则表达式还会在Perl中的另外两个地方出现。split函数的第一个参数是一个特殊的匹配操作符，指定将一个字符串分解为多个子串时哪些部分不用返回。参见第27章有关split的描述和例子。qr//（“引用表达式”）操作符也通过一个正则表达式指定模式，不过它不做任何匹配（这与m//不同，m//要完成匹配）。相反，它会返回正则表达式的编译形式，供将来使用。有关的更多信息请参见后面的“变量内插”一节。



要用`=~`绑定操作符（这并不是一个真正的操作符，从语言学角度讲，这只是一种主题工具）对一个特定的字符串应用`m///`、`s///`或`tr///`操作符。下面给出几个例子：

```
$haystack =~ m/needle/      # 匹配一个简单模式
$haystack =~ /needle/       # 同样的工作
$italiano =~ s/butter/olive oil/ # 合法的替换
$rotate13 =~ tr/a-zA-Z/n-za-mN-ZA-M/ # 很容易破解的简单加密
```

如果没有绑定操作符，`$_`只用作“主题”：

```
/new life/ and      # 在$_中搜索（如果找到）
/new civilizations/ # 再次搜索$_

s/sugar/aspartame/   # 完成$_替换

tr/ATCG/TAGC/        # 补全$_中的DNA
```

由于`s///`和`tr///`会改变它们处理的标量，所以只能对合法的左值使用这些操作符：<sup>注3</sup>

```
"onshore" =~ s/on/off/; # 不正确：编译时错误
```

不过，可以对任何标量表达式的结果应用`m///`：

```
if ((lc $magic_hat->fetch_contents->as_string) =~ /rabbit/) {
    say "Nyaa, what's up doc?";
}
else {
    say "That trick never works!";
}
```

但是必须特别当心，因为`=~`和`!~`的优先级相当高，在前一个例子中，必须为左边的项加括号<sup>注4</sup>。`!~`绑定操作符的工作与`=~`类似，不过它将操作的逻辑结果取反：

```
if ($song !~ /words/) {
    say qq/"$song" appears to be a song without words./;
}
```

由于`m///`、`s///`和`tr///`都是引号操作符，你可以选择你自己的定界符。它们与引号操作符`q///`、`qq///`、`qr///`和`qw///`做法相同（参见第2章中“选择你自己的引号”一节）。

```
$path =~ s#/tmp#/var/tmp/scratch#;

if ($dir =~ m[/bin]) {
    say "No binary directories please.";
}
```

对`s///`或`tr///`使用成对的定界符时，如果第一部分是4种常用ASCII括号对之一（尖括号、圆括号、中括号或花括号），可以为第二部分选择与第一部分不同的定界符：

---

注3：除非使用`/r`修饰符将修改后的结果作为一个左值返回。

注4：如果没有括号，低优先级的`lc`会应用到整个模式匹配，而不只是`magic hat`对象上的调用。



```
s(egg)<larva>;
s{larva}{pupa};
s[pupa]/imago/;
```

允许在开始定界符前面加空白符：

```
s (egg) <larva>;
s {larva} {pupa};
s [pupa] /imago/;
```

每次一个模式成功匹配时，它会把\$`、\$&和\$'变量设置为匹配的左文本、整个匹配和匹配的右文本。这对于将字符串分解为不同分量很有用：

```
"hot cross buns" =~ /cross/;
say "Matched: <$`> $& <$'>"; # Matched: <hot > cross < buns>
say "Left: <$`>"; # Left: <hot >
say "Match: <$&>"; # Match: <cross>
say "Right: <$'>"; # Right: < buns>
```

为了得到更好的粒度和效率，可以使用小括号捕获你想保留的特定部分。每一对括号会捕获与括号中子模式对应的子串。括号对从左到右按左括号的位置编号；匹配之后，可以通过编号变量\$1、\$2、\$3等得到与这些子模式对应的子串：<sup>注5</sup>

```
$_ = "Bilbo Baggins's birthday is September 22";
/((.*)'s birthday is (.*)/);
say "Person: $1";
say "Date: $2";
```

\$`、\$&、\$'和编号变量都是全局变量，隐式地局部化到外围动态作用域。它们会一直保留到下一次成功的模式匹配，或者到达当前作用域的末尾（要看哪一个先出现）。有关的更多内容将在后面介绍，我们会在一个不同的作用域中讨论。

一旦Perl发现你在程序中的某个地方需要用到\$`、\$&或\$'（这3个变量中的任何一个），它就会为每一个模式匹配提供这些变量。这会让程序稍稍变慢。Perl生成\$1、\$2等变量时也使用了一个类似的机制，所以对于包含捕获小括号的模式，你也要付出一定开销（参见这一章后面“无捕获的分组”一节，无捕获的分组在保留分组行为的同时可以避免捕获的开销）。不过，如果没有使用\$`、\$&或\$'，不包含捕获小括号的模式就不存在性能开销。所以，通常最好尽可能避免\$`、\$&和\$'，特别是在库模块中。不过，如果必须使用一次（有些算法确实很喜欢这些变量的便利），那就尽管随心使用吧，因为你已经为之付出了开销，使用一次和使用多次的开销都是一样的。在最近的Perl版本中，\$&没有另外两个变量的开销大。

一种更好的选择是/p匹配修饰符（将在之后讨论）。它会保留匹配的字符串，所以

---

注5： 不过，没有\$0，\$0中保存的是程序名。

`${^PREMATCH}`、`${^MATCH}`和`${^POSTMATCH}`变量包含了原本`$``、`$&`和`$'`要包含的内容，不过不会影响整个程序的性能。

## 模式修饰符

稍后我们会讨论各个模式匹配操作符，不过首先需要提到它们共有的另外一个特性：修饰符（modifier）。

紧挨着`m///`、`s///`、`qr///`、`y///`或`tr///`操作符的最后一个定界符，可以在后面以任意顺序放置一个或多个单字母修饰符（可选）。为简洁起见，修饰符通常写为“`/i modifier`”，读作“斜线i修饰符”，尽管最后一个定界符可能不是斜线（有时人们会用“标志”或“选项”表示“修饰符”，都是可以的）。

有些修饰符会改变单个操作符的行为，所以后面我们还会详细介绍这些修饰符。另外一些修饰符会改变解释正则表达式的方式，这些修饰符将在这里介绍。`m///`、`s///`和`qr///`操作符<sup>注6</sup>的最后一个定界符后面接受以下修饰符，见表5-1。

表5-1：正则表达式修饰符

修饰符	含义
<code>/i</code>	忽略字母大小写（不区分大小写）
<code>/s</code>	使 <code>.</code> 也能匹配换行符
<code>/m</code>	使 <code>^</code> 和 <code>\$</code> 也能匹配邻近嵌入的 <code>\n</code>
<code>/x</code>	忽略（大多数）空白符，允许模式中有注释
<code>/o</code>	模式只编译一次
<code>/p</code>	保留 <code>\${^PREMATCH}</code> 、 <code>\${^MATCH}</code> 和 <code>\${^POSTMATCH}</code> 变量
<code>/d</code>	ASCII–Unicode双模式字符集行为（原来的默认行为）
<code>/a</code>	ASCII字符集行为
<code>/u</code>	Unicode字符集行为（新的默认行为）
<code>/l</code>	运行时本地化环境的字符集行为（有 <code>use locale</code> 声明时的默认行为）

`/i`修饰符要求以任何可能的大小写匹配一个字符；也就是说，匹配时不考虑大小写，这个过程也称为大小写转换（casefolding）。这表示不仅匹配大写和小写，还要匹配标题形式（titlecase）的字符（英语中未用）。如果同一个字符的同一个大小写有多个变种，就需要这种不区分大小写的匹配，比如希腊sigma字符就有两个小写形式：大写字母“Σ”的小写通常是“σ”，不过在词尾时会变成“ς”。例如，希腊语Σίσυφος（这就是我们熟知的“西绪福斯”（Sisyphus））中同时包含sigma的这三种形式。

由于不区分大小写的匹配要根据字符来匹配，而不是根据语言匹配<sup>注7</sup>，它会匹配另

注6： `tr///`操作符不接收正则表达式，所以这些修饰符不适用。

注7： 嗯，几乎是这样。不过我们确实不想讨论语言问题，所以干脆说不是按语言匹配。



一种语言中可能认为大小写有错误的文本。所以/perl/i不仅匹配“perl”，还能匹配类似“proPErly”或“perLiter”等字符串，这些并不是真正的英语。类似的，希腊语/σίυφος/i不仅匹配“ΣΊΣΥΦΟΣ”和“Σίυφος”，还能匹配难看的“ςίυφοο”，两端换上了两个小写sigma。

这是因为，即使我们声称某些字符串是英语或希腊语，但Perl并不知道。它只是采用一种不区分语言的方式完成它的大小写无关匹配。由于同一个字母的所有大小写形式都有同样的大小写转换（casefold），所以它们都能匹配。

由于Perl只支持8位本地化环境（locale），低于256的本地化环境匹配（locale-matching）码点会使用当前的本地化环境映射来确定大小写转换，不过更大的码点则要使用Unicode规则。在本地化环境中，不区分大小写的匹配不能超过255/256边界，而且可能还有其他限制。

/s和/m修饰符没有任何特异之处。它们只影响Perl以何种方式匹配一个包含换行符的字符串。不过并不是判断你的字符串是否确实包含换行符；它们关心的是Perl应该认为字符串包含一行（/s）还是多行（/m），这是因为，取决于是否采用面向行的方式，某些元字符会做不同的处理。

正常情况下，元字符“.”可以匹配除了换行符以外的任何单个字符，因为它的传统含义就是匹配一行中的字符。不过，如果有/s，“.”元字符还可以匹配一个换行符，因为通过/s，你已经告诉Perl要忽略字符串可能包含多个换行符这一事实。如果你希望即使有/s也不匹配换行符，可以使用\N，这与[^\\n]的含义相同，只是输入更容易。

另一方面，/m修饰符会改变^和\$元字符的解释，使它们也能匹配字符串中邻近的换行符，而不只是字符串末尾的换行符（/m会禁用优化，认为你在匹配一行，所以不要随意使用这个修饰符）。请参见这一章后面“位置”一节中的例子。

/p修饰符将匹配本身的文本保留在特殊变量\${^MATCH}中，匹配前的文本保留在\${^PREMATCH}中，而匹配后的文本保留在\${^POSTMATCH}中。

现在/o修饰符基本上已经过时了，它能控制模式的再编译。如今，这个修饰符放在长度在10kB以上的模式前才会有效果，所以这已经算是一种“老古董”了。不过仍有可能在老代码中看到这个修饰符，所以下面还会介绍这个修饰符是如何工作的。

除非选择的定界符是单引号（m'PATTERN'、s'PATTERN'REPLACEMENT'或qr'PATTERN'），否则每次计算模式操作符时，模式中的所有变量都会正常地内插。最坏情况下，这可能导致模式的重新编译。即使在最好的情况下，这也会带来一个字符串比较，查看是否需要重新编译。如果希望这个模式编译一次且仅一次，可以使用/o修饰符。这会避免开销巨大的运行时再编译。如果内插的值在执行期间不会改变，这就很有用。不过，如果使用了/o就



意味着做出了承诺：你不会改变模式中的变量。如果确实改变了变量，Perl甚至不会注意到。为了更好地控制重新编译，应当使用`qr//`正则表达式引号操作符。有关的详细内容请参见这一章后面的“变量内插”一节。

`/x`是表述性修饰符：允许你利用空白符和解释性注释来提高模式的可读性，甚至可以扩展为多行模式，也就是可以越过换行符边界。

嗯，这就是说，`/x`修饰符会修改空白符（以及`#`字符）的含义：使它们不再像普通字符那样与自身匹配，而是把它们变成元字符（有意思的是，现在它们会表现出空白符（和注释字符）应有的行为）。因此，`/x`允许用空格、制表符和换行符指定格式，就像常规的Perl代码一样。它还允许模式中使用并无特殊含义的`#`字符引入一个注释，这个注释可以延伸到模式串中当前行的末尾<sup>注8</sup>。如果你想匹配一个真正的空白符（或`#`字符），必须把它放在一个中括号字符类中，或者使用一个八进制或十六进制转义对其编码（不过空白符通常与`\s*`或`\s+`序列匹配，所以在实际中这种情况不常出现）。

总之，在这些特性的大力帮助下，使得传统的正则表达式成为一种更可读的语言。本着TMTOWTDI的精神，如今编写一个给定的正则表达式可以有不只一种方法。实际上，甚至不只两种方法：

```
m/\w+:(\s+\w+)\s*\d+;/      # 单词，冒号，空格，单词，空格，数字。
```

```
m/\w+: (\s+ \w+) \s* \d+/x;  # 单词，冒号，空格，单词，空格，数字
```

```
m{
    \w+:          # 匹配一个单词和一个冒号
    (            # (开始捕获组)
        \s+      # 匹配一个或多个空格
        \w+      # 匹配另一个单词
    )            # (结束捕获组)
    \s*          # 匹配0个或多个空格
    \d+          # 匹配一些数字
}x;
```

我们会在这一章后面解释这些新的元符号（这一节主要讨论模式修饰符，不过因为我们对`/x`太过着迷，所以这里有些跑题了）。下面的正则表达式能查找段落中的重复单词，这是从《Perl Cookbook》中借用来的。其中使用了`/x`和`/i`修饰符，另外还用到了后面将要介绍的`/g`修饰符。

```
# 找出段落中重复的单词，可能会跨行边界。
# 使用/x匹配空格和注释，
# 用/i匹配"Is is this ok?"中的两个'is'，并使用/g找出所有重复
$/ = "";          # "paragrep" 模式
while (<>) {
    while ( m{
```

---

注8： 要当心不要在注释中包含模式定界符。由于Perl的“先查找末尾”原则，它并不知道你本意不是在这里终止模式。

```

        \b          # 从单词边界开始
        (\w\S+)    # 找到一个像单词的块
        (
            \s+      # 用某个空白符分隔
            \1        # 然后又是一个块
        ) +        # 任意重复
        \b          # 直到另一个单词边界
    }xig
)
{
    say "dup word '$1' at paragraph $.";
}
}

```

对本文本文（英文版）运行这个脚本时，会得到类似下面的警告：

```
dup word 'that' at paragraph 150
```

实际上，我们是故意这样写的（重复双写“that”）。

`/u`修饰符会为模式匹配启用Unicode语义。如果模式在内部用UTF-8编码，或者在一个`use feature "unicode_strings" pragma`作用域中编译（除非也在原来的`use locale`或`use bytes pragma`作用域中编译，这两个`pragma`已经过时了），就会自动设置这个修饰符。

如果有`/u`修饰符，码点128~255（也就是介于128和255之间，包括128和255）取其ISO-8859-1 (Latin-1)含义，这与Unicode含义相同。如果没有`/u`，在一个非UTF-8字符串上使用`\w`只能完全匹配`[A-Za-z0-9_]`，而不能与其他模式匹配。如果有`/u`，对一个非UTF-8字符串使用`\w`则能匹配128~255的所有Latin-1单词字符。具体来讲，就是MICRO SIGN  $\mu$ 、两个序数指示符<sup>9</sup>和<sup>0</sup>，以及62个拉丁字母（在UTF-8字符串中，`\w`能与所有这些匹配）。

`/a`修饰符会改变`\d`、`\s`、`\w`和POSIX字符类，使之只与ASCII范围内的码点匹配<sup>注9</sup>。这些序列正常情况下会与Unicode码点匹配，而不只是ASCII。如果有`/a`，`\d`只表示10个ASCII数字“0”到“9”，`\s`只表示5个ASCII空白符`[\f\n\r\t]`，`\w`只表示63个ASCII单词字符`[A-Za-z0-9_]`（这也会影响`\b`和`\B`，因为它们是按`\w`转换定义的）。类似地，所有POSIX类（如`[[:print:]]`）只在有`/a`的情况下匹配ASCII字符。

也许你没想到，在某个方面，`/a`更像是`/u`：它不保证ASCII字符只匹配ASCII。例如，基于Unicode大小写转换规则，“S”、“s”和“ſ”（U+017F LATIN SMALL LETTER LONG S）能以不区分大小写的方式相互匹配，“K”、“k”和U+212A KELVIN SIGN, (“K”)也是如此。只需要将这个修饰符写两次，变成`/aa.`，就能禁用这种有趣的Unicode大小写转换。

`/l`修饰符要求模式匹配时使用当前本地化环境的规则。这里的“当前本地化环境”是指执行匹配时起作用的本地化环境，而不是编译时的本地化环境。在支持本地化环境的系统

---

注9： 在这个平台上谈到ASCII时，还在使用EBCDIC的人应该适当地改换一下想法了。Perl的在线文档对EBCDIC码有更详细的讨论。



上，可以使用POSIX模块的setlocale函数改变当前本地化环境。对于在一个“use locale”pragma作用域中编译的模式而言，这个修饰符是缺省的。

Perl只支持单字节本地化环境，而不支持多字节的本地化环境。这说明，大于255的码点会处理为Unicode，而不论哪个本地化环境起作用。按照Unicode规则，不区分大小写的匹配可以跨过255和256之间的单字节边界，不过这在/l下必须禁用。

这是因为，本地化环境中对字符的码点赋值与在Unicode下是不同的（除了真正的ISO-8859-1以外）。因此，本地化字符255并不能以不区分大小写的方式与字符376（U+0178 LATIN CAPITAL LETTER Y WITH DIAERESIS (ÿ)）匹配，因为在当前本地化环境中，255可能不是U+00FF LATIN SMALL LETTER Y (y)。Perl甚至无法知道这个字符串是否在本地化环境中存在，更无法知道它的码点。

如果显式要求Perl使用v5.14特性集，则默认有/u修饰符。如果不是这样，现有的代码仍会像以前一样工作，就好像在各个模式上使用了一个/d修饰符一样（或者如果有use locale声明，则像是使用了/l）。这可以确保向后兼容性，同时还为将来完成工作提供了一种更简洁的方法。传统的Perl模式匹配行为有二元性，因此/d也可以表示“看情况而定”（“it depends”）。如果有/d，Perl会根据平台的本地字符集完成匹配，除非有其他情况指示应当使用Unicode规则。这些情况包括：

- 目标字符串或模式本身内部采用UTF-8编码。
- 大于255的码点。
- 使用\p{PROP}或\P{PROP}指定的属性。
- 用\N{NAME}指定的命名字符、别名或序列，或者使用\N{U+HEXDIGITS}指定的码点。

如果没有任何声明要求/u、/a或/l语义，则假设为双元模式/d。有/d的模式可能有Unicode行为，也可能没有。历史上，ASCII和Unicode语义总是混杂在一起，而这也引发了无休止的争论，所以使用v5.14时这不再是默认模式。或者你也可以显式地改为更直观的Unicode模式。以下声明都可以启用Unicode字符串：

```
use feature "unicode_strings";
use feature ":5.14";
use v5.14;
use 5.14.0;
```

也可以使用与以上4个pragma对应的命令行选项来启用Unicode字符串：

```
% perl -Mfeature=unicode_strings more arguments
% perl -Mfeature=:5.14 more arguments
% perl -M5.014 more arguments
% perl -M5.14.0 more arguments
```



由于-E命令行选项表示使用当前版本的特性集，所以这也能启用Unicode字符串（v5.14及以上版本中）：

```
% perl -E code to eval
```

与大多数pragma一样，还可以对各个作用域禁用特性，所以下面这个pragma：

```
no feature "unicode_strings";
```

会禁用可能在外围语法作用域中声明的Unicode字符集语义。

为了能更容易地控制正则表达式行为，而不是每次都增加同样的模式修饰符，现在可以使用re pragma来设置或清除一个词法作用域中的默认标志。

```
# 为所有模式设置默认修饰符
use re "/msx"; # 作用域中的模式会增加这些修饰符

# 现在为一个内部作用域清除一些修饰符
{
    no re "/ms"; # 这个作用域中的修饰符去掉了这些修饰符
    ...
}
```

这对于与字符集行为有关的模式修饰符尤其有用：

```
use re "/u";      # Unicode 模式
use re "/d";      # ASCII-Unicode双工模式
use re "/l";      # 8位本地化环境模式
use re "/a";      # ASCII模式，加上Unicode大小写转换
use re "/aa";     # ASCII模式，没有Unicode大小写转换
```

利用这些声明，你就不用为了得到一致的语义（甚至是一致的错误语义）而多次重复。

## m//操作符（匹配）

```
m/PATTERN/modifiers
/PATTERN/modifiers
?PATTERN?modifiers (过时)

EXPR =~ m/PATTERN/modifiers
EXPR =~ /PATTERN/modifiers
EXPR =~ ?PATTERN?modifiers (过时)
```

m//操作符在标量EXPR中搜索与PATTERN匹配的字符串。如果使用/或?作为定界符，最前面的m是可选的。?和'作为定界符有特殊的含义：前者(?)是一个单次匹配，后者(')会禁止变量内插和6个转义转义（\U等，见后面的介绍）。

如果PATTERN计算为一个空串（这可能是因为你用//指定了空串，或者是因为一个内插的变量计算为空串），就会使用未被一个内部块（或者split、grep或map）隐藏的最后一个成功执行的正则表达式。

在标量上下文中，如果匹配成功，则这个操作符返回true (1)，否则返回false (""）。这种形式常见于布尔上下文：

```
if ($shire =~ m/Baggins/) { ... }      # 在$shire中搜索Baggins
if ($shire =~ /Baggins/) { ... }      # 在$shire中搜索Baggins

if ( m#Baggins# ) { ... }             # 在$_中搜索
if ( /Baggins/ ) { ... }              # 在$_中搜索
```

在列表上下文中使用时，m//返回与模式中捕获小括号（即\$1、\$2、\$3等）匹配的一个子串列表，见“分组和捕获”一节的介绍。即使返回列表，仍会设置编号变量。如果在列表上下文中匹配失败，会返回一个空列表。如果在列表上下文中匹配成功，但是没有捕获小括号（也没有/g），则返回一个值为(1)的列表。由于失败时返回一个空列表，所以这种形式的m//也可以用在布尔上下文中，不过只能通过一个列表赋值间接参与：

```
if (($key,$value) = /(\\w+): (\\.*)/) { ... }
```

m//的合法修饰符（各种形式）如表5-2所示。

表5-2：m//修饰符

修饰符	含义
/i	忽略字母大小写（不区分大小写）
/m	使^和\$也能匹配邻近嵌入的\\n
/s	使.也能匹配换行符
/x	忽略（大多数）空白符，允许模式中有注释
/o	模式只编译一次
/p	保留匹配的字符串
/d	ASCII–Unicode双模式字符集行为（原来的默认行为）
/u	Unicode字符集行为（新的默认行为）
/a	ASCII字符集行为
/l	运行时本地化环境的字符集行为（ 有use locale声明时的默认行为）
/g	全局查找所有匹配
/cg	允许在失败的匹配后继续搜索

这些修饰符大多应用于模式，我们在前面已经讨论过。最后两个会改变匹配操作本身的行为。/g修饰符指定全局匹配。也就是说，在字符串中匹配尽可能多的次数。具体行为取决于上下文。在列表上下文中，m//g会返回所找到的所有匹配的一个列表。这里我们会找到所有提到“perl”、“Perl”、“PERL”等的地方：

```
if (@perls = $paragraph =~ /perl/gi) {
    printf "Perl mentioned %d times.\\n", scalar @perls;
}
```

如果/g模式中没有捕获小括号，就会返回完整的匹配。如果有捕获小括号，则只返回捕获的字符串。假设有一个类似这样的字符串：

```
$string = "password=xyzyzzy verbose=9 score=0";
```

另外假设你想用它初始化一个散列，如下：

```
%hash = (password => "xyzyzzy", verbose => 9, score => 0);
```

当然，现在还没有列表，只有一个字符串。为了得到相应的列表，可以在列表上下文中使用m//g操作符从这个字符串中捕获所有键/值对：

```
%hash = $string =~ /(\w+)= (\w+)/g;
```

(\w+)序列会捕获一个字母数字单词。参见下一节“分组和捕获”。

如果在标量上下文中使用，/g指示一个渐进式匹配，这会让Perl在同一个变量上从上一次匹配后面的位置开始下一次匹配。\G断言表示字符串中的这个位置。参见这一章后面的“位置”一节，其中对\G有详细的介绍。除了/g，如果还使用了/c修饰符（表示“继续”），那么/g处理结束后，即使匹配失败，也不会重置位置指针。

如果定界符是一个?，如m?PATTERN?（或?PATTERN?，不过没有m的版本已经过时了），这就像一个正常的/PATTERN/搜索，只不过它在reset操作符调用之间只匹配一次。如果你希望程序运行期间只匹配模式的第一次出现，而不是所有出现，这将是一个很方便的优化方法。每次调用时，这个操作符会运行搜索，直到最后匹配到某个结果，在此之后它会自行关闭，并返回false，直到你显式地用reset再将它打开。Perl会跟踪匹配状态。

如果一个普通的模式匹配想找出最后一次出现而不是第一次出现，这种情况下m??操作符最有用：

```
open(DICT, "/usr/dict/words") or die "Can't open words: $!\n";
while (<DICT>) {
    $first = $1 if m? (^ neur .* ) ?x;
    $last = $1 if m/ (^ neur .* ) /x;
}
say $first;          # 打印"neurad"
say $last;           # 打印"neuropnology"
```

reset操作符只重置与reset调用所在同一个包中编译的那些??实例。m??就等价于??。

## s///操作符（替换）

```
s/PATTERN/REPLACEMENT/modifiers
```

```
LVALUE =~ s/PATTERN/REPLACEMENT/modifiers
```

```
RVALUE =~ s/PATTERN/REPLACEMENT/rmodifiers
```



这个操作符会搜索与*PATTERN*匹配的一个字符串，如果找到，将这个匹配的子串替换为*REPLACEMENT*文本。如果*PATTERN*是一个空串，则使用最后一个成功执行的正则表达式。

```
$lotr = $hobbit;          # 只复制The Hobbit
$lotr =~ s/Bilbo/Frodo/g; # 很容易地写结局
```

如果使用*/r*修饰符，*s///*操作的返回值就是结果字符串，目标字符串不做任何改变。如果没有*/r*修饰符，*s///*操作的返回值（在标量上下文以及列表上下文中）则是它成功替换的次数。如前所述，如果结合使用了*/g*修饰符，可能成功不只一次。如果失败，由于它替换了0次，所以会返回false（""），这在数值上等价于0<sup>注10</sup>。

```
if ($lotr =~ s/Bilbo/Frodo/) { say "Successfully wrote sequel." }
$change_count = $lotr =~ s/Bilbo/Frodo/g;
```

正常情况下，每次替换时与*PATTERN*匹配的所有内容都会被丢掉，不过可以在模式中包含*\K*来“保留”一部分：

```
$tales_of_Rohan =~ s/Éo\Kmer/wyn/g; # 重写历史
```

替换部分被当作一个双引号字符串。可以在替换字符串中使用之前介绍的任何动态作用域模式变量（*\$`*，*\$&*，*\$'*，*\$1*，*\$2*等），另外还可以使用你喜欢的任何其他双引号模式。例如，下面这个例子会查找字符串“revision”、“version”或“release”，把它们分别替换为相应的大写形式，这里在替换部分使用了*\u*转义序列：

```
s/revision|version|release/\u$&/g; # 模式中使用|表示“或”
```

所有标量变量都会在双引号上下文中展开，而不只是这些特殊的变量。假设有一个*%Names*散列，它将版本号映射到内部的项目名。例如，*\$Names{"3.0"}*可能是名为“Isengard”的代码。可以使用*s///*找到版本号，把它们替换为相应的项目名：

```
s/version ([0-9.]+)/the $Names{$1} release/g;
```

在替换字符串中，*\$1*返回捕获的第一个（且唯一一个）小括号对（如果愿意，也可以在模式中使用*\1*，不过在替换中这种用法已经过时。在一个正常的双引号字符串中，*\1*表示一个Control-A）。

*PATTERN*和*REPLACEMENT*都要完成变量内插，不过*PATTERN*是在每次作为一个整体计算*s///*操作符时完成内插，而*REPLACEMENT*是在每次模式匹配时进行内插（如果使用*/g*修饰符，一次计算中*PATTERN*可以匹配多次）。

与前面一样，表5-3中的修饰符大多要改变正则表达式的行为，它们与*m//*和*qr//*的修饰符是一样的。后面3个修饰符会改变替换操作符本身。

---

注10：与*m//*操作符和第3章介绍的更多传统操作符一样，这是一个特殊的false值，可以安全地用作数字。这是因为，与一个普通的空串（null串）不同，这个“”隐式转换为数字时不会有数值警告。

表5-3: s///修饰符

修饰符	含义
/i	忽略字母大小写（匹配时）
/m	使^和\$也能匹配邻近嵌入的\n
/s	使.也能匹配换行符
/x	忽略（大多数）空白符，允许模式中有注释
/o	模式只编译一次
/p	保留匹配的字符串
/d	ASCII-Unicode双模式字符集行为（原来的默认行为）
/u	Unicode字符集行为（新的默认行为）
/a	ASCII字符集行为
/l	运行时本地化环境的字符集行为（有 <i>use locale</i> 声明时的默认行为）
/g	全局替换，也就是说，替换所有出现
/cg	允许在失败的匹配后继续搜索
/r	返回替换，原字符串保持不变
/e	将右边作为一个表达式来计算

/g修饰符用于s///时，可以将PATTERN的所有匹配替换为REPLACEMENT值，而不只是所找到的第一个匹配。s///g操作符相当于一个全局搜索和替换操作符，所有改变同时发生，即使在标量上下文中也不例外（这与m//g不同，它是渐进式的）。

/r（非破坏性）修饰符将替换应用到字符串的一个新副本上，现在不再要求这个字符串是一个变量。不论替换是否发生，都会返回这个副本，原来的字符串总保持不变：

```
say "Déagol's ring!" =~ s/D/Sm/r;      # 打印"Sméagol's ring!"
```

这个副本往往是一个普通字符串（即使输入是一个对象或是一个绑定变量）。这个修饰符最早出现在v5.14生产版本中。

/e修饰符把REPLACEMENT处理为一个Perl代码块，而不是作为一个内插的字符串。执行这个代码得到的结果将用作替换字符串。例如，s/([0-9]+)/sprintf("%#x", \$1)/ge将把所有数字转换为十六进制，比如，2581会转换为0xb23。或者在前一个例子中，假设你不确定是否已经掌握所有版本的名字，对于那些不知道名字的版本，你可能希望它们保持不变。利用一些创造性的/x格式化，可以写为：

```
s{
    version
    \s+
    (
        [0-9.]+
    )
}
```

```

    $Names{$1}
    ? "the $Names{$1} release"
    : $&
}xge;

```

`s///e`的右边（在这里实际上是“下边”）会在编译时连同程序的其他部分一起完成语法检查和编译。所有语法错误将在编译期间检测，而运行时异常并不捕获。第一个`/e`后面每多一个`/e`（如`/ee`、`/eee`等）等价于对代码的结果再调用`eval STRING`，每多一个`/e`就会调用一次。这会计算代码表达式的结果，并把异常捕获到特殊变量`$@`中。有关的更多细节请参见这一章后面的“编程化模式”一节。

## 顺带修改字符串

有时你希望基于一个老字符串得到一个修改后的新字符串，但不要改变原来的老字符串。可以不这样写：

```

$lotr = $hobbit;
$lotr =~ s/Bilbo/Frodo/g;

```

可以将它们合并到一个语句中。由于优先级的原因，这里需要对赋值加小括号，因为`=~`应用到表达式时有最高的优先级。

```

($lotr = $hobbit) =~ s/Bilbo/Frodo/g;

```

如果赋值两边没有加小括号，那么只会改变`$hobbit`，并把替换个数存储到`$lotr`中，这个结果很怪异。

没错，在更新的代码中，你可以直接使用`/r`：

```

$lotr = $hobbit =~ s/Bilbo/Frodo/gr;

```

不过很多Perl程序员还在使用原来的惯用做法。

## 顺带修改数组

不能对数组直接使用`s///`操作符。要修改数组，需要一个循环。很巧合的是，利用`for/foreach`的别名机制，再加上使用`$_`作为默认循环变量，就可以得到一个标准的Perl惯用做法，能够搜索和替换数组中的各个元素：

```

for (@chapters) { s/Bilbo/Frodo/g }    # 逐章完成替换
s/Bilbo/Frodo/g for @chapters;         # 同样的处理

```

与处理简单的标量变量一样，如果想保留原来的值不变，则可以把替换和赋值结合起来：

```

@oldhues = ("bluebird", "bluegrass", "bluefish", "the blues");
for (@newhues = @oldhues) { s/blue/red/ }
say "@newhues"; # 打印: redbird redgrass redfish the reds

```



要达到同样的目的，另一种方法是结合使用/r替换修饰符（v5.14新增）和map操作：

```
@newhues = map { s/blue/red/r } @oldhues;
```

要对同一个变量完成重复替换，惯用的方法是使用一个“直通的”循环。例如，下面展示了如何规范化一个变量中的空白符：

```
for ($string) {  
    s/^\s+//;      # 丢弃前导空白符  
    s/\s+$//;      # 丢弃末尾空白符  
    s/\s+/ /g;     # 合并内部空白符  
}
```

这与以下代码生成的结果相同：

```
$string = join(" ", split " ", $string);
```

还可以在这样一个循环中结合使用赋值，就像前面的数组例子中一样：

```
for ($newshow = $oldshow) {  
    s/Fred/Homer/g;  
    s/Wilma/Marge/g;  
    s/Pebbles/Lisa/g;  
    s/Dino/Bart/g;  
}
```

## 如果全局替换不够全局

有时不能使用/g来得到所有修改（替换），可能是因为替换重叠或者必须从右向左替换，或者可能因为在匹配之间\$`的长度要改变。通过反复调用s///总能达到目的。不过，你希望s///最后失败时循环停止，所以必须把它放在条件中，这样一来，循环的主要部分就没有工作可做了。所以我们只写一个1，这很无聊，不过无聊还算是好的，有时可能还很怪异。下面给出几个例子，其中使用了更多奇怪的正则表达式模式：

```
# 在整数的适当位置加逗号  
1 while s/(\d)(\d\d\d)(?!\\d)/$1,$2/;  
  
# 将制表符扩展为8列空格  
1 while s/\\t+/" " x (length($&)*8 - length($`)%8)/e;  
  
# 删除（嵌套的（甚至深层嵌套（就像这样）））的标志  
1 while s/\\([^(\\)]*)\\//g;  
  
# 删除重复出现两次的单词（以及三次（和四次…））  
1 while s/\\b(\\w+) \\1\\b/$1/gi;
```

最后一个需要循环，因为如果没有循环，它会把以下字符串：

```
Paris in THE THE THE THE spring.
```

转换成：

Paris in THE THE spring.

这可能会让懂一点法语的人觉得巴黎位于一个喷冰茶的喷泉中，因为“thé”在法语中就表示“茶”。当然，巴黎人可不会被骗。

## tr///操作符（转换）

```
tr/SEARCHLIST/REPLACEMENTLIST/cdsr
```

```
LVALUE =~ tr/SEARCHLIST/REPLACEMENTLIST/cds
RVALUE =~ tr/SEARCHLIST/REPLACEMENTLIST/cdsr
RVALUE =~ tr/SEARCHLIST//c
```

对于喜欢*sed*的人来说，*y///*可以作为*tr///*的一个同义词。正因如此，不能调用一个名为*y*的函数，同样地，也不能调用名为*q*或*m*的函数。在所有其他方面，*y///*都与*tr///*相同，这一点我们不再重复。

这个操作符本来不该出现在关于模式匹配的一章中，因为它没有使用模式。这个操作符会逐字符地扫描一个字符串，并把*SEARCHLIST*（这不是一个正则表达式）中找到的一个字符的每一次出现替换为*REPLACEMENTLIST*（这不是一个替换字符串）中的相应字符。不过，这看起来与*m//*和*s///*有些相似，甚至可以对它使用`=~`或`!~`绑定操作符，所以我们仍在这一章介绍这个操作符（*qr//*和*split*都是模式匹配操作符，不过不能结合使用绑定操作符，所以我们将其他章节中介绍这两个操作符。你可以自己找找看）。

转换操作会返回替换或删除的字符个数。如果没有通过`=~`或`!~`操作符指定字符串，则会修改`$_`字符串。*SEARCHLIST*和*REPLACEMENTLIST*可以用一个短横线来定义顺序字符范围：

```
$message =~ tr/A-Za-z/N-ZA-Mn-za-m/; # rot13加密
```

注意，类似A-Z的范围会假设这是一个线性字符集，如ASCII。不过每个字符集都有自己的排序方式，相应地对于哪些字符落入一个特定的范围，不同字符集也有不同的想法。一个合理的原则是，只使用以相同大小写字母开头和结尾的范围（a-e，A-E）或开头和结尾都是数字的范围（0~4）。其他范围都有问题。如果还有疑问，可以把字符全部拼出来：ABCDE。甚至简单的[A-E]都会失败，而[ABCDE]可以正常使用，因为拉丁语中小体大写字母的码点很分散，参见表5-4。

表5-4：小体大写字母及其码点

文字	码点	类别	脚本	名
A	U+1D00	GC=Ll	SC=Latin	LATIN LETTER SMALL CAPITAL A
B	U+0299	GC=Ll	SC=Latin	LATIN LETTER SMALL CAPITAL B
C	U+1D04	GC=Ll	SC=Latin	LATIN LETTER SMALL CAPITAL C
D	U+1D05	GC=Ll	SC=Latin	LATIN LETTER SMALL CAPITAL D
E	U+1D07	GC=Ll	SC=Latin	LATIN LETTER SMALL CAPITAL E

*SEARCHLIST*和*REPLACEMENTLIST*不会像双引号字符串那样完成变量内插。不过，可以使用映射到特定字符的反斜线序列，如\n或\015。

表5-5列出了tr///操作符可用的修饰符。这些修饰符与应用于m//、s///或qr//的修饰符完全不同，尽管看上去一样。

表5-5：tr///修饰符

修饰符	含义
/c	求 <i>SEARCHLIST</i> 的补集
/d	删除找到但是未替换的字符
/s	合并压缩重复的替换字符
/r	返回转换字符串，原字符串保持不变

如果使用了/r修饰符，转换是在字符串的一个新副本上完成的，而且最后返回的也是这个副本。不要求它是一个左值。

```
say "Drogo" =~ tr/Dg/Fd/r; # Drogo -> Frodo
```

如果指定了/c修饰符，*SEARCHLIST*中设置的字符会求补。也就是说，实际的搜索列表由所有不在*SEARCHLIST*中的字符组成。对于Unicode，这可能表示相当多的字符，不过由于它们是逻辑存储的，而不是物理存储，所以不用担心耗尽内存。

/d修饰符把tr///变成所谓的“转换”操作符：由*SEARCHLIST*指定但在*REPLACEMENTLIST*中未给出替换的所有字符都将被删除（这比一些tr(1)程序的-d行为要稍稍灵活一些，后者会简单地删除在*SEARCHLIST*中找到的所有字符）。

如果指定了/s修饰符，则可以转换为相同字符的字符序列将会合并压缩，最终转换为这个字符的一个实例。

如果使用了/d修饰符，*REPLACEMENTLIST*总会按指定的方式解释。否则，如果*REPLACEMENTLIST*比*SEARCHLIST*短，会重复最后一个字符，直到这个串足够长。如果*REPLACEMENTLIST*为null，则会复制*SEARCHLIST*，如果你只是想统计字符数而不是要改变这些字符，这会非常有用。还可以用/s压缩字符。如果只统计字符，则可以使用任何右值（*RVALUE*），而不只是左值（*LVALUE*）。

```
tr/aeiou!/!;          # 将所有元音字符改为!
tr{/\\\r\n\b\f. }{ _}; #将奇怪字符更改为下划线

$count = ($para =~ tr/\n//); # 统计$para中的换行符数
$count = tr/0-9//;         # 统计$_中的数字

tr/@$%*//d;           # 删除这些字符

# 顺带修改
```



```
($HOST = $host) =~ tr/a-z/A-Z/;

# 最终结果相同，不过作为右值
$HOST = ($host =~ tr/a-z/A-Z/r);

$pathname =~ tr/a-zA-Z/_/cs; # 将除ASCII字母外的所有其他字符
                             # 改为一个下划线
```

如果同一个字符在`SEARCHLIST`中出现多次，只使用第一次出现的字符。因此：

```
tr/AAA/XYZ/
```

将把所有单个字符A改为X（在`$_`中）。

尽管变量不会内插到`tr///`，但通过使用`eval EXPR`仍能得到同样的效果：

```
$count = eval "tr/$oldlist/$newlist/";
die if $@; # 传播非法eval内容产生的异常
```

还要说明一点：如果你想把文本改为大写或小写，不要使用`tr///`。可以在双引号字符串中使用`\U`或`\L`序列（或者等价的`uc`和`lc`函数），因为它们会关注本地化环境或Unicode信息，而`tr/a-zA-Z/`不会。另外，在Unicode字符串中，`\u`序列以及相应的`ucfirst`函数知道标题形式（`titlecase`）的概念，对于某些字符，标题形式绝不是简单地转换为大写。

`\F`序列对应于`fc`函数，参见第27章中对`fc`的介绍。这是v5.16新增的特性，用于简单的大小写无关比较，如`"\F$a" eq "\F$b"`或等价的`fc($a) eq fc($b)`。`/i`修饰符总是在大小写无关的匹配中用来在内部完成大小写转换。`\F`和`fc`现在更容易访问。参见第6章中“Unicode文本比较和排序”一节。

## 元字符和元符号

既然已经了解了所有这些千奇百怪的笼子，下面再来看看关在这些笼子里的“动物”，也就是放在模式中的奇怪符号。现在你已经知道一个事实：这些符号并不是函数调用或算术操作符那样的常规Perl代码。正则表达式本身就是一种语言，这是一个嵌套在Perl中的小语言（就像我们这个社会也包含着丛林）。

为了实现其强大的能力和表现力，Perl中的模式能够识别同样的12个传统元字符（就像是十二金刚），很多其他正则表达式包中也可以看到这些元字符：

```
\ | ( ) [ { ^ $ * + ? .
```

其中一些元字符会改变规则，使它后面原本正常的字符有特殊的表现。我们不想把这些长序列称为“字符”，所以构成长序列时，我们称之为元符号（`metasymbols`，或者有时也直接称为“符号”）。不过，在顶层你（和Perl）只需要考虑这12个元字符。所有其他的模式都是由此开始的。

一些简单的元字符就表示它们自身，如`^`和`$`。它们不会直接影响旁边的字符。有些元字符就像是前缀操作符，会控制它们后面的内容，如`\`。另外一些则像是后缀操作符，控制着紧挨在它们前面的部分，如`*`、`+`和`?`。还有一个元字符类似于一个中缀操作符，位于它控制的两个操作数之间。甚至还有一些括号元字符，类似于环缀操作符，控制着其中包含的内容，如`(...)`和`[...]`。小括号尤其重要，因为它们在内部指定了`|`的边界，在外部指定了`*`、`+`和`?`的边界。

如果在这12个元字符中选择，只学习其中最重要的一个，那么应该选反斜线（嗯，还有小括号）。这是因为，反斜线会禁用其他元字符。在一个Perl模式中，如果反斜线放在一个非字母数字字符前面，它会让下一个字符变成一个直接量。如果在模式中需要按字面匹配这12个元字符，就要在它们前面加一个反斜线。因此，`\.`会匹配一个真正的点号，`\$`匹配一个真正的美元符，`\\`匹配一个真正的反斜线，依此类推。这称为对元字符“转义”，或“引用”，有时也直接称为“加反斜线”（当然，你已经知道反斜线可以用来禁止双引号字符串中的变量内插）。

尽管反斜线会把一个元字符变成一个直接量字符，它对于后面的字母数字字符的影响则完全不同。它会把原本正常的东西变得特殊。也就是说，与字母数字字符一起会形成一个元符号。表5-3给出了这些元符号的一个列表（按字母顺序排序）。

## 元符号表

在下面几个表中，如果给定的元符号是可量化的（能匹配有宽度的东西），相应的“原子性”一列就会写为“是”。另外，我们会用“...”表示“其他”（如果从表中简单的一行解释不能了解它的含义，请看后面的讨论）。

表5-6给出了基本的传统元符号。前面4个是之前提到的结构元符号，后面3个是简单的元字符。`.`元字符是一个原子，因为它可以匹配有宽度的东西（在这里，宽度为一个字符）；`^`和`$`是断言，因为它们匹配0宽度的东西，这些字符的计算只是为了查看它们是否为true。

表5-6：通用正则表达式元字符

符号	原子性	含义
<code>\...</code>	可变	使下一个字母数字字符为元字符，或者（可能）使下一个非字母数字字符不作为元字符
<code>... ...</code>	否	候选（匹配一个或另一个）
<code>(...)</code>	是	分组（当作一个单元）
<code>[...]</code>	是	字符类（匹配一组中的一个字符）
<code>^</code>	否	如果在字符串开头（或者可能在某个换行符后面）则为true
<code>.</code>	是	匹配一个字符（正常情况下除换行符以外）
<code>\$</code>	No	如果在字符串末尾（或者可能在某个换行符前面）则为true



量词会在另外一节中更深入地介绍，指示了前面的原子（即单个字符或分组）应当匹配多少次，如表5-7所列。

表5-7：正则表达式量词

最大	最小	占有	允许范围
{MIN,MAX}	{MIN,MAX}?	{MIN,MAX}?+	必须出现至少MIN次，但是不超过MAX次
{MIN,}	{MIN,}?	{MIN,}?+	必须出现至少MIN次
{COUNT}	{COUNT}?	{COUNT}?+	必须正好匹配COUNT次
*	*?	*+	0或多次（等同于{0,}
+	+	++	1或多次（等同于{1,}
?	??	?+	0或1次（等同于{0,1}

最小量词会在其允许范围内匹配尽可能少的字符。最大量词会在其允许范围内尝试匹配尽可能多的字符。

例如，.+保证匹配字符串中的至少一个字符，但是如果提供了机会，它会匹配所有字符。这些机会将在这一章后面“小引擎/能（与不能）/”一节中介绍。

占有量词有些像最大量词，只不过它在回溯期间不会放弃任何已经匹配的结果，而最小和最大量词在回溯时可能改变匹配的数量。

要注意量词不可以量化。像??和++之类本身就是量词，分别是最小量词和占有量词，而不是对自身量化的普通单字符量词。只有标志为原子时才能量化，而量词不是原子。

我们希望提供一个可扩展的语法以便构造新的元符号。由于只需要处理12个元字符，我们选择了一种之前不合法的正则表达式序列，用来表示任意的语法扩展。除了最后一个元符号，这些元符号都采用(?KEY...)形式，也就是说，一个（平衡的）小括号后面有一个问号，再后面是一个KEY和子模式的其余部分。KEY字符指示这是哪一个特定的正则表达式扩展。表5-8给出了这些扩展的正则表达式序列。其中大部分都反映在结构上，因为它们都基于小括号，不过还有附加的含义。重申一次，只有原子可以量化，因为它们表示（可能）真正存在的东西。

表5-8：扩展的正则表达式序列

扩展	原子性	含义
(?#...)	否	注释，丢弃
(?:...)	是	非捕获组
(?>...)	是	占有组，不捕获也不回溯
(?adlupimsx-imsx)	否	启用/禁用模式修饰符
(?^alupimsx)	否	重置和启用模式修饰符



表5-8：扩展的正则表达式序列（续）

扩展	原子性	含义
(?adlupimsx-imsx:...)	是	只用于分组的小括号，以及启用/禁用修饰符
(?^alupimsx:...)	是	只用于分组的小括号，以及重置和启用修饰符
(?=...)	否	如果前瞻（lookahead）断言成功则为true
(?!...)	否	如果前瞻（lookahead）断言失败则为true
(?<=...)	否	如果后顾（lookbehind）断言成功则为true
(?<!...)	否	如果后顾（lookbehind）断言失败则为true
(? ... ... ...)	是	为编号分组重置分支
(?<NAME>...)	是	命名捕获分组；也记为(?'NAME'...)。参见下面的 \k<NAME>
(?{...})	否	执行嵌入的Perl代码
(??{...})	是	匹配嵌入Perl代码中的正则表达式
(?NUMBER)	是	调用组NUMBER中的独立子表达式；也记作(?+NUMBER)，(?-NUMBER)，(?0)和(?R)。要确保这里不能使用&字符
(?&NAME)	是	在组NAME上递归；确保这里一定要用&字符。另外可以写作(?P>NAME)
(?(COND)... ...)	是	匹配if-then-else模式
(?(COND)...)	是	匹配if-then模式
(?(DEFINE)...)	否	为完成后面的“正则表达式子例程”调用，将命名组定义为(?&NAME)
(*VERB)	否	回溯控制动词，也写作 (*VERB:NAME)

回溯控制动词还处于试验阶段，所以这里不做讨论。不过，如果你的工作涉及向导工具，可能会经常遇到这种回溯控制动词。所以如果看到以下动词，请查看Perlre手册页：

```
(*ACCEPT)
(*COMMIT)
(*FAIL)      (*F)
(*MARK:NAME) (*:NAME)
(*PRUNE)     (*PRUNE:NAME)
(*SKIP)      (*SKIP:NAME)
(*THEN)      (*THEN:NAME)
```

或者也可以直接运行。

最后，表5-9显示了你喜欢的所有字母数字元符号（对于在变量内插阶段处理的符号，相应地在“原子性”列中有一个短横线标志，因为引擎永远不会看到这些符号）。

表5-9：字母数字正则表达式元符号

符号	原子性	含义
\0	是	匹配字符数字0 (U+0000, NULL, NUL)
\ NNN	是	匹配用八进制给定的字符，最大为\377
\ n	是	匹配第n个捕获组（十进制）
\a	是	匹配警报字符 (ALERT, BEL)
\A	否	在字符串开头时为true
\b	是	匹配退格字符 (BACKSPACE, BS)（只在字符类中）
\b	否	在单词边界时为true
\B	否	不在单词边界时为true
\c X	是	匹配控制字符 Control-X (\cZ, \c[ 等)
\C	是	匹配一字节 (C char)，甚至包括采用UTF-8格式的字节（危险！）
\d	是	匹配任何数字字符
\D	是	匹配任何非数字字符
\e	是	匹配转义字符 (ESCAPE, ESC,，非反斜线)
\E	—	结束大小写(\F, \L, \U)或反斜线(\Q)转换
\f	是	匹配换页字符 (FORM FEED, FF)
\F	—	转换为Foldcase（而不是小写），直到\E结束 <sup>a</sup>
\g{GROUP}	是	匹配命名或编号捕获组
\G	否	在前一个m//g的匹配结束位置时为true
\h	是	匹配所有水平空白符
\H	是	匹配除水平空白符以外的所有字符
\k<GROUP>	是	匹配命名捕获组；也写作\k'NAME'
\K	否	使\K左边的文本不匹配
\l	—	只将下一个字符小写（而不是foldcase）
\L	—	小写（而不是foldcase），直到\E结束
\n	是	匹配换行符（通常为LINE FEED, LF）
\N	是	匹配除换行符以外的所有字符
\N{NAME}	是	匹配命名字符、别名或序列，如\N{greek:Sigma}匹配“Σ”
\o{NNNN}	是	匹配用八进制给定的字符
\p{PROP}	是	匹配任何有命名属性的字符
\P{PROP}	是	匹配任何没有命名属性的字符
\Q	—	引用（消元）元字符，直到\E结束
\r	是	匹配回车字符（通常是CARRIAGE RETURN, CR）
\R	是	匹配任何换行字形簇（不在字符类中）
\s	是	匹配任何空白符
\S	是	匹配任何非空白符
\t	是	匹配制表符 (CHARACTER TABULATION, HT)
\u	—	只是下一个字符转换为Titlecase（不是大写）



表5-9：字母数字正则表达式元符号（续）

符号	原子性	含义
\U	—	转换为大写（而不是titlecase），直到\E结束
\v	是	匹配任何垂直空白符
\V	是	匹配除垂直空白符以外的任何字符
\w	是	匹配任何“单词”字符（字母、数字、组合标记和连接符号）
\W	是	匹配任何非单词字符
\x{abcd}	是	匹配用十六进制给定的字符
\X	是	匹配字形簇（不在字符类中）
\z	否	只在字符串末尾为true
\Z	否	在字符串末尾或可选的换行符前面时为true

a: \V和相应的fc函数是v5.16新增的特性。

如果属性名是一个字符，\p和\P上的大括号是可选的。如果十六进制数为2位或小于2位，\x上的大括号则是可选的。没有大括号的\N表示一个非换行符，而不是一个命名字符。如果引用的捕获组是数字，\g上的大括号是可选的（不过最好还是使用大括号）。

\R匹配一个CARRIAGE RETURN后面跟一个LINE FEED（占有性），或者是任何一个垂直空白符。它等价于(?>\r\n|\v)。占有组意味着"\r\n" =~ /\R\n/永远不会匹配；一旦看到两字符CRLF，以后将不会再把它改为单独的CARRIAGE RETURN，即使模式后面需要LINE FEED才能使整个模式成功，也不会拆开CRLF。

只有描述为“匹配…”或“匹配任何…”的元符号可以用在字符类（中括号）中，而且要求它们只匹配一个字符，所以\R和\X是不允许的。也就是说，字符类一次只能匹配一个字符，所以它们仅限于包含单个字符的特定集合；在这些字符类中，只能使用描述其他单字符特定集合的元符号，或者描述特定单个字符的元符号。当然，这些元符号也可以与所有其他非类元符号一起在字符类之外使用。不过，要注意\b可以“分身”为两个完全不同的家伙：在字符类中它是退格字符，而在字符类之外它又是一个单词边界断言。

\K（为便于记忆，可以理解为：“保持”（Keep）你已经匹配的结果）不匹配任何东西。实际上，它会告诉引擎重置它珍藏的某个东西（匹配的某一部分），如\$&或\${^MATCH}变量，或者替换的左边。参见s///操作符中的例子。

模式可以匹配的字符与正常双引号字符串可以内插的字符之间有一定的重叠。由于正则表达式要经过两轮处理，一个给定字符会由哪一轮来处理，有时这是不明确的。如果不明确，变量内插阶段会延迟这些字符的解释，而留给正则表达式解析器来处理。

不过只有知道正在解析一个正则表达式时，变量内插阶段才会把这些字符的解释交给正则表达式解析器处理。可以将正则表达式指定为普通的双引号字符串，不过这样一来，就必



须遵循双引号规则。之前能够映射到具体字符的元符号仍能正常匹配，尽管它们并没有交给正则表达式解析器来处理。不过在正常的双引号字符串中不能使用任何其他元符号（实际上在任何双引号上下文，如``...``、`qq(...)`、`qx(...)`或内插的`here`文档中都不允许使用那些元符号）。如果希望将字符串解析为一个正则表达式而不做任何匹配（暂且不匹配），应当使用`qr//`（引用正则表达式）操作符。

另一方面，大小写和反斜线转换转义（`\u`等）必须在变量内插阶段处理，因为这些元符号的目的就是影响变量如何内插。如果用单引号禁止变量内插，那么也不会得到转换转义。变量和转换转义（`\u`等）都不会将在单引号字符串中展开，也不会单引号`m'...'`或`qr'...'`操作符中展开。即使确实会完成内插，如果这些转换转义作为变量内插的结果出现，也会被忽略，因为那时要影响变量内插已经为时太晚了。

尽管转换操作符不接受正则表达式，不过我们讨论过的所有匹配单个特定字符的元符号都可以在`tr///`操作中使用。其余的元符号则不行（除了反斜线以外，它仍能以通常的方式起作用）。

## 特定字符

前面已经提到，模式中非特殊的东西都会匹配自身。这说明`/a/`会匹配“a”，`/=/`匹配“=”等。不过，有些字符可不容易输入，即使你想办法输入了这些字符，它们会把你的屏幕格式搅得一团糟（比如说你“有幸”用到了控制字符，它们就因失控而臭名昭著）。为了解决这个问题，正则表达式会识别双引号引起的字符别名，如表5-10所示。

表5-10：双引号字符别名

转义	含义
<code>\0</code>	Null字符 (NUL, NULL)
<code>\a</code>	警报 (BEL, ALERT)
<code>\e</code>	转义 (ESC, ESCAPE)
<code>\f</code>	换页 (FF, FORM FEED)
<code>\n</code>	换行 (LF, LINE FEED)
<code>\r</code>	回车 (CR, CARRIAGE RETURN)
<code>\t</code>	制表符 (HT, HORIZONTAL TAB)

与双引号字符串中一样，模式还支持以下5个元符号：

`\cX`

这是一个命名ASCII控制字符，如`\cC`表示Control-C、`\cZ`表示Control-Z、`\c[`表示ESC，`\c?`表示DEL。相应的序数值必须是0~31或127。

## `\NNN`

使用2位或3位八进制码指定的字符。除了小于010（十进制的8）的值外，前导0是可选的，因为（与双引号字符串中不同），一位数都认为都是由模式中编号捕获组捕获的字符串引用。如果在模式之前至少捕获了 $n$ 个子串（这里 $n$ 是一个十进制数），多位数会被解释为第 $n$ 个引用。否则，它们将解释为用八进制码指定的一个字符。

## `\x{HEXDIGITS}`

由一位或两位十六进制数（[0-9a-fA-F]）指定的码点（字符编号），如`\x1B`。只有当后面跟着的字符不是一个十六进制数时才可以使用一位数形式。如果使用了大括号，可以根据需要使用多位数字。例如，`\x{262f}`会匹配一个Unicode U+262F YIN YANG (☯)。

## `\N{NAME}`

一个命名字符、别名或序列，如`\N{GREEK SMALL LETTER EPSILON}`、`\N{greek:epsilon}`或`\N{epsilon}`。这需要有第29章介绍的`chardnames pragma`，它还会确定可能采用哪种形式使用这些名字（`":full"`对应于以上给出的第一种形式，`":short"`对应另外两种形式）。

还可以使用`\N{U+NUMBER}`记法指定字符。例如，`\N{U+263B}`表示☹，即BLACK SMILING FACE字符。这种用法并不需要`chardnames pragma`。

你喜欢的任何Unicode标准文档中都可以找到所有Unicode字符的一个列表，或者可以通过迭代处理`chardnames::via code(N)`（ $N$ 从0到0x10\_FFFF）来生成所有Unicode字符，记住要跳过surrogate字符。

## `\o{NUMBER}`

这是用八进制码指定的一个字符。与有二义性的`\NNN`记法不同，这可以是任意多位八进制数，而且不会与捕获引用混淆。

# 通配元符号

有3种特殊的元符号可以用作通用的通配符，它们分别匹配“任意”字符（“任意”有特定的含义）。这些通配符分别为点号（`.`）、`\C`和`\X`。它们都不能用在中括号字符类中。字符类中不能使用点号，这是因为点号（几乎）匹配存在的任意字符，所以本身就是一个通用字符。如果想包含或排除一切，则没有必要再使用一个中括号字符类。特殊通配符`\C`和`\X`有特殊的结构化含义，但与选择单个Unicode字符的概念不太一致，中括号字符类适用于选择单个Unicode字符。

点号元字符匹配除换行符以外的任何字符（如果有`/s`修饰符，换行符也能匹配。在这种情况下，可以使用`\N`匹配非换行符）。与模式中的12个特殊字符类似，要从字面上匹配一个点号，必须用一个反斜线将它转义。例如，下面的代码会检查一个文件名是否以一个点号后面跟着一个单字符扩展名结尾：



```
if ($pathname =~ /\.(.)\z/s) {
    say "Ends in $1";
}
```

第一个点号，即被转义的点号，是直接量字符，而第二个表示“匹配任何字符”。\z指出只在字符串末尾匹配，/s修饰符允许点号还能匹配换行符（没错，使用换行符作为文件扩展名确实不怎么样，不过这并不表示不能这么做）。

点号元字符经常与量词结合使用。.\*会匹配尽可能多的字符，而.\*?匹配尽可能少的字符。不过有时也可以单独使用而不用量词来指定它的宽度：/(...):(...):(...)/匹配3个冒号分隔的域，这3个域分别为两字符长。

不要把字符与字节混淆了。以前，点号只匹配单字节，而现在它会匹配Unicode字符，很多Unicode字符无法编码为单字节：

```
use charnames qw[ :full ];
$BWV[887] = "G\N{MUSIC SHARP SIGN} minor";
my ($note, $black, $mode) = $BWV[887] =~ /^([A-G])(.)\s+(\S+)/;
say "That's lookin' sharp!" if $black eq chr 0x266f; ##
```

\X元符号以一种更扩展的方式匹配一个字符。它匹配由一个或多个Unicode字符构成的字符串，称为“字形簇”（grapheme cluster）。这表示获取连续的多个字符，它们共同为用户显示一个字形。一般地，字形簇有一个基字符，后面会结合一些附加符号，如变音符或分音符，这些附加符号与基字符共同构成一个逻辑单元。还可以是任何Unicode换行序列，包括"\r\n"，由于没有对换行应用标记，这甚至可以是位于字符串开头或行开头的一个单独的标记。

Perl原来的\X的做法非常类似于(?\PM\pM\*)>，不过表现并不太好，所以Unicode改进了它的字形簇的概念。字形簇的准确定义很复杂，不过以下定义已经很接近了：

```
(?> \R
| \p{Grapheme_Base} \p{Grapheme_Extend}*
| \p{Grapheme_Extend}
)
```

重点是\X匹配一个用户可见的字符（字形），尽管这可能是程序员可见的多个字符（码点）。如果以上伪扩展中的\R匹配一个CRLF对，或者如果字形基字符后面是一个或多个字形扩展字符<sup>注11</sup>，/\X/匹配的字符串的长度可能超过一个字符。占有组表示一旦找到一个基字符以及后面的任意多个扩展字符，\X就不能再改变心意。例如，/\X.\z/不能匹配“cafe\x{301}”，这里U+0301是COMBINING ACUTE ACCENT，因为\X不能回溯。

---

注11：通常是组合标记。当前只有以下非标记字形扩展字符：ZERO WIDTH NONJOINER、ZERO WIDTH JOINER、HALFWIDTH KATAKANA VOICED SOUND MARK和HALFWIDTH KATAKANA SEMI-VOICED SOUND MARK。



如果使用Unicode，而且确实想得到一字节而不是一个字符，可以使用\c元符号。这会匹配一字节（具体的，一个C语言char类型），即使这可能与你的Unicode字符流不一致。如果这样做，可能会有一些警告，有关内容参见第6章。不过，要达到这个目的（即匹配一字节），这可能并不是合适的方法。实际上，可以使用Encode模块将字符串解码为字节（也就是说，码点小于256的字符）。

## 字符类

在模式匹配中，可以匹配任何有（或者没有）某个特定属性的字符。指定字符类有4种方法。可以用传统方式指定一个字符类，也就是使用中括号，并在其中枚举所有可能的字符，或者可以使用3种助记的快捷方式：经典Perl类（如\w）、使用属性（如\b{word}，或者使用遗留的POSIX类（如[:word:]）。这些快捷方式只匹配相应字符集合中的一个字符。将它们量化（加量词）可以匹配更大的扩展字符串，如\d+可以匹配一个或多个数字（可能很容易犯这样一个错误，认为\w会匹配一个单词。实际上，使用\w+才能匹配一个单词，这里所说的“单词”是指一个可以包含下划线和数字等的编程语言标识符，而不是英语中的一个单词）。

## 中括号字符类

中括号中的一个枚举字符列表称为一个中括号字符类（bracketed character class），它只匹配这个列表中的一个字符。例如，[aeiouy]会匹配英语中的一个元音字母。要匹配一个真正的中括号，可以加反斜线，或者把它放在列表最前面。

字符范围可以使用一个连字符<sup>注12</sup>和a-z记法来指示。可以结合多个范围，例如，[0-9a-fA-F]可以匹配一个十六进制“数字”。可以使用一个反斜线来保护连字符，否则这会被解释为一个范围分隔符，或者可以把它放在类的开头或结尾（这种做法可读性不太好，但这是一种比较传统的做法）。

如果把脱字符号（或抑扬符，也叫做帽子符或上箭头）放在中括号字符类前面，可以将字符类取反，使它匹配任何不在列表中的单个字符（要匹配一个脱字符号，就不要把它放在最前面，或者更好的办法是用一个反斜线将其转义）。例如，[^aeiouy]会匹配不是元音字母的所有其他字符。不过对于字符类取反要特别当心，因为字符的全集在不断扩展。例如，中括号字符类会匹配辅音字符，另外还匹配空格、换行符和古代斯拉夫语、希腊语或几乎所有其他脚本中的所有字符（包括元音），更不用说中文、日语和韩语中的表意文字。也许有一天还能匹配Cirth和Tengwar精灵文（古希腊的B类线形文字和意大利的伊特鲁里亚文字肯定能匹配）。所以最好明确地指定你的辅音字符，如

---

注12：实际上，由U+002D, HYPHEN-MINUS 指示，而不是U+2010, HYPHEN。

[cbdfghjklmnpqrstvwxyz]或简写为[b-df-hj-np-tv-z]。这样还能直接解决两个地方（元音字符类和辅音字符类）中都出现“y”的问题，这个问题无法用补集来解决。

字符类支持表示一个特定字符的普通字符元符号，如\n、\t、\cX、\xNN、\NNN（表示八进制数，而不是反向引用）、\p{YESPROP}和\N{NAME}。另外，还可以在字符类中使用\b表示一个退格，就像在双引号字符串中一样。正常情况下，在模式匹配中这表示一个单词边界。不过0宽度断言在字符类中没有任何意义，所以在这里\b回归了它在字符串中的正常含义。任何单个字符都可以用作一个范围的终点，不论是用作一个直接量字符、一个经典的反斜线转义（如\t）、作为其十六进制或八进制码点，还是使用命名字符。

字符类还支持所有表示特定字符集的元符号，包括取反字符类如\P{NOPROP}、\N、\S和\D，以及本章后面将要介绍的预定义字符类（经典字符类、Unicode字符类或POSIX字符类）。不过，不要将它们用作范围的终点，这是没有意义的，所以“-”要从字面上解释。另外使用可能超过1个字符长的符号也是没有意义的。但\R和\X除外，因为\R能同时匹配一个回车和一个换行，而\X能匹配连续的多个码点，或者通过\N{NAME}匹配某个能扩展为多个码点的命名序列。

所有其他元符号都会在中括号中失去其特殊含义。具体地，3个通用的通配符（“.”、\X或\C）都不能使用。不能使用“.”通常让人很惊讶，不过在一个受限的字符类中使用通用字符类没有太大意义，通常你只是想匹配一个点号直接量作为字符类的一部分。例如，匹配文件名时就需要匹配点号直接量。另外在一个中括号字符类中指定量词、断言或候选项也是没意义的，因为字符会单独解释。例如，[fee|fie|foe|foo]与[feio|]的含义相同。

中括号字符类通常只与一个字符匹配。由于这个原因，v5.14中的Unicode名序列不能用于中括号字符类中（或者不能像你希望地那样使用）。它们看上去像命名字符，不过实际上长度为多个字符。例如，LATIN CAPITAL LETTER A WITH MACRON AND GRAVE可以在\N{...}构造中使用，不过它实际上会扩展为U+0100和后面跟着的U+0300。在中括号内，这个命名序列就是[\x{100}\x{300}]，这可能不是你想要的。

不过，如果有/i，中括号字符类有时可以匹配多个字符。这是因为，通过完全大小写转换，字符串中的一个字符可以匹配模式中的多个符号，反之亦然。例如，下面的代码就是成立的：

```
"SS" =~ /\^[ \xDF]$/iu
```

这是因为U+00DF的大小写转换是“ss”，而“SS”的大小写转换也是“ss”。由于大小写转换相同，所以匹配成功。不过，在取反的字符类中（如[^\xDF]），完全大小写转换会降级为简单的大小写转换，因为如果不降级就会导致逻辑矛盾。只有在这种情况下Perl会使用简单的大小写转换；正常情况下，Perl中的所有大小写转换和大小写映射都是完全转换，而不是简单转换。



# 经典Perl字符类简写

从一开始，Perl就提供了很多字符类简写。如表5-11所列。所有这些都是加反斜线的字母元符号，另外，对于每个元符号，其大写版本就是将小写版本取反。

这些字符类匹配的字符比你所认为的多得多，因为这通常会作用于整个Unicode范围，而不只是ASCII（对于取反的字符类，甚至会超出Unicode的范围）。不管怎样，正常的含义是ASCII或本地化含义的一个超集。关于属性和遗留的POSIX形式，更多解释见这一章后面“POSIX风格字符”一节。如果要保留原来的ASCII含义，要对该范围使用use re “/a”声明，或者在各个模式上加一个或两个/a。

表5-11：经典字符类

符号	含义	正常属性	/a属性	/a枚举	遗留 [:POSIX:]
\d	数字	\p{X_POSIX_Digit}	\p{POSIX_Digit}	[0-9]	[:digit:]
\D	非数字	\P{X_POSIX_Digit}	\P{POSIX_Digit}	[^0-9]	[:^digit:]
\w	单词字符	\p{X_POSIX_Word}	\p{POSIX_Word}	[_A-Za-z0-9]	[:word:]
\W	非单词字符	\P{X_POSIX_Word}	\P{POSIX_Word}	[^_A-Za-z0-9]	[:^word:]
\s	空白符	\p{X_Perl_Space}	\p{Perl_Space}	[\t\n\f\r]	[:space:] <sup>a</sup>
\S	非空白符	\P{X_Perl_Space}	\P{Perl_Space}	[^\t\n\f\r]	[:^space:]
\h	水平空白符	\p{Horiz_Space}	\p{Horiz_Space}	很多	[:blank:]
\H	非水平空白符	\P{Horiz_Space}	\P{Horiz_Space}	很多	[:^blank:]
\v	垂直空白符字符	\p{Vert_Space}	\p{Vert_Space}	很多	—
\V	非垂直空白符字符	\P{Vert_Space}	\P{Vert_Space}	很多	—

a：不过不包括VTAB。

没错，我们知道大多数单词中都不包括数字或下划线；\w是按典型编程语言（或者，更具体地讲就是Perl）中的token来匹配“单词”。

这些元符号可以在中括号内使用，也可以在中括号外使用。也就是说，可以单独使用，也可以作为某个构造的中括号字符类的一部分：

```
if ($var =~ /\D/) { warn "contains a nondigit" }
if ($var =~ /[^\w\s.]/) { warn "contains non-(word, space, dot)" }
```

这些元符号的定义大多遵循Unicode标准。尽管Perl在内部使用Unicode，不过还有很多老程序，这些老程序并不知道这一点，这会带来一些意外。所以Perl中的传统字符类缩写都存在某种多重人格问题，有时它们表示一种东西，而有时却表示另一种东西。如果有/u标



志，这种双重人格就会消失，字符串总是有Unicode语义。由于这才是正确的道路，所以在有`use v5.14`或更好的声明时，`/u`标志是缺省的（`unicode_strings`特性也会设置这个标志为缺省标志）。

由于传统的原因，`\s`与`[\h\v]`不同，因为`\v`包括`\cK`，这是一个很少使用的垂直制表符。正是因为这个原因，Perl的`\s`不能完全等于Unicode的`\p{Whitespace}`属性。

如果使用遗留的本地化环境（有`use locale`或`use re "/l"`），对于小于256的码点会得到这些字符的本地化环境含义，而对于256及更大的码点，会得到其正常含义。

对于大于255的码点，正常情况下Perl会切换为一个纯字符解释。这说明类似U+0389的码点（GREEK CAPITAL LETTER OMEGA）总会解释为一个`\w`字符。

不过，如果有`/a`或`/aa`修饰符，就不再是这样。通常，对于Unicode存在之前设计的那些老模式，人们会使用这种仅用于ASCII的修饰符来强制只按ASCII语义解释这些老模式。不用为需要这个修饰符的每一个模式都加一个`/a`，可以使用以下词法作用域`pragma`，有了这个声明就会自动认为有`/a`：

```
use re "/a";
```

这会排除一些字符，如空白符。这还表示ISO-8859-1中的非ASCII字母对于`\w`字符不再算是字母。

## 字符属性

可以使用`\p{PROP}`和它的补集`\P{PROP}`得到字符属性。对于7种主要的Unicode通用类别（Unicode General Category）属性（名字里只有一个字母），`\p`和`\P`的大括号是可选的。所以可以写为`\pL`表示任何字母，或者写为`\pN`对应所有数字，不过更长的属性（包括多个字母）则必须使用大括号，如`\p{Lm}`或`\p{Nl}`。

大多数属性都在Unicode标准中直接定义，不过有些属性往往由这些标准属性组合而成，它们是Perl特有的属性。例如`Nl`和`Mn`是标准Unicode通用类别，分别表示字母数字和非空格组合标记，而`Perl_Space`是Perl自己的属性。

属性可以单独使用，也可以在一个构造字符类中结合使用：

```
if ($var =~ /^[\p{alpha}]+$/) { say "all alphabetic" }
if ($var =~ s/[\p{L}\p{N}]/g) { say "deleted all nonalphanumerics" }
```

属性有很多，其中一些常用的属性分别涵盖多个字符，实际上它们所包括的字符数比很多人想象的多得多。例如，`alpha`和`word`属性就分别涵盖100 000个字符，`word`相对来讲更大一些，因为它是`alpha`的一个超集。

可以在perluniprops手册页中找到你的Perl版本所支持的当前属性列表，其中还会列出每个属性匹配多少个字符。Perl密切跟踪着Unicode标准，所以随着Unicode中增加新的属性，这些新属性也会增加到Perl中。官方的Unicode属性参见UAX#44: Unicode字符数据库，以及UTS #18: Unicode正则表达式Annex C的兼容属性。如果所有这些还不够，甚至还可以定义你自己的属性，参见第6章来了解如何做到。

最常用的属性就是Unicode通用类别属性。表5-12显示了所有这7个单字符类别，还包括它们的长形式和含义。

表5-12: Unicode通用类别（主要类别）

短属性	长属性	含义
C	Other	错误控制码等
L	Letter	字母和表意文字
M	Mark	组合标记
N	Number	数字
P	Punctuation	标点符号
S	Symbol	符号、标志和印记
Z	Separator	分隔符

这7个属性实际上分别是以该字母开头的两字符通用类别属性的别名。表5-13给出了所有通用类别属性的完备集合。所有字符（甚至包括当前未指定的字符）都属于以下的某一个通用类别。

表5-13: Unicode通用类别（全部类别）

短名	长名	含义
Cc	Control	ASCII和Latin-1的C0和C1控制码
Cf	Format	特殊文本中的不可见字符
Cn	Unassigned	尚未指定字符的码点
Co	Private Use	指定你自己的含义
Cs	Surrogate	为UTF-16保留的非字符
Ll	Lowercase_Letter	小写字母
Lm	Modifier_Letter	上标字母和间隔附加符号
Lo	Other_Letter	单独字母和表意文字
Lt	Titlecase_Letter	仅首字母大写，如一个句子的第一个单词
Lu	Uppercase_Letter	大写字母，全大写文本中使用的大写字母
Mc	Spacing_Mark	占据一个打印列的很小的结合部分
Me	Enclosing_Mark	包围另一个字符的组合标记
Mn	Nonspacing_Mark	不占打印列的很小的结合部分



表5-13: Unicode通用类别（全部类别）（续）

短名	长名	含义
Nd	Decimal_Number	一位数字，表示大端十进制数中使用的0~9
Nl	Letter_Number	字母用作数字，如罗马数字
No	Other_Number	任何其他类型的数字，如分数
Pc	Connector_Punctuation	连接符号，如下划线
Pd	Dash_Punctuation	任何类型的短横线或连字符（但不是减号）
Pe	Close_Punctuation	类似结束括号的标点符号
Pf	Final_Punctuation	类似右引号的标点符号
Pi	Initial_Punctuation	类似左引号的标点符号
Po	Other_Punctuation	所有其他标点符号
Ps	Open_Punctuation	类似开始括号的标点符号
Sc	Currency_Symbol	货币中使用的符号
Sk	Modifier_Symbol	主要是附加符号
Sm	Math_Symbol	数学中使用的符号
So	Other_Symbol	所有其他符号
Zl	Line_Separator	U+2028
Zp	Paragraph_Separator	U+2029
Zs	Space_Separator	所有其他非控制空白符

所有标准Unicode属性实际上由两部分组成，如`\p{NAME=VALUE}`。因此，所有单部分属性都是对官方Unicode属性的补充。值为true的布尔属性都可以缩写为单部分，所以可以写为`\p{Lowercase}`表示`\p{Lowercase=True}`。除了布尔属性外，其他类型的属性可以取字符串、数值或枚举值。Perl还为所有通用类别、文字系统和块属性提供了单部分别名，另外根据Unicode关于正则表达式的技术标准#18（13版，2008年08月发布）提供了一级推荐，如`\p{Any}`。

例如，`\p{Armenian}`、`\p{IsArmenian}`和`\p{Script=Armenian}`都表示同样的属性，`\p{Lu}`、`\p{GC=Lu}`、`\p{Uppercase_Letter}`和`\p{General_Category=Uppercase_Letter}`也表示同一个属性。二进制属性（值隐含为true）还有另外一些例子，包括`\p{Whitespace}`、`\p{Alphabetic}`、`\p{Math}`和`\p{Dash}`。不是二进制属性的例子包括`\p{Bidi_Class=Right_to_Left}`、`\p{Word_Break=A_Letter}`和`\p{Numeric_Value=10}`。*perluniprops*手册页列出了所有属性以及Perl支持的相应别名，包括标准Unicode属性和Perl特殊属性。

如果将一个非Unicode码点（即大于0x10FFFF）与Unicode属性匹配，结果将是未定义的。目前，默认地这会产生一个警告，匹配失败。有些情况下，这违反我们的直觉，因为下面两个匹配都会失败：



```

chr(0x110000) =~ \p{ahex=true}      # false
chr(0x110000) =~ \p{ahex=false}     # false!

chr(0x110000) =~ \P{ahex=true}      # true
chr(0x110000) =~ \P{ahex=false}     # true!

```

不过，用户自定义的属性可以有自己的行为。参见第6章的“构建字符”一节。

## POSIX风格字符类

与Perl的其他字符类简写不同，遗留的POSIX风格字符类语法记法`[ :CLASS: ]`仅在构造其他字符类时才可用，也就是说，只能用在附加的一对中括号中。例如，`/[.,[:alpha:][:digit:]]/`会搜索一个字符，这可以是一个直接量点号（因为它在中括号字符类中）、逗号、字母字符或者数字。所有这些都可以用作同名的字符属性，例如 `[.,\p{alpha}\p{digit}]`。

除了“punct”外（稍后就会解释），POSIX字符类名可以在有相同含义的`\p{}`或`\P{}`中用作属性。这有两个优点：这样更容易输入，因为不需要在它们两边加上额外的大括号；另外可能更重要的是：作为属性，它们的定义不再受字符集修饰符的影响，而总是作为Unicode匹配。与之相反，如果使用`[[:...:]]`记法，则POSIX类会受修饰符标志的影响。

`\p{punct}`属性与`[[:punct:]]` POSIX类不同，因为`\p{punct}`不会匹配非标点符号，而`[[:punct:]]`（以及`\p{POSIX_Punct}`和`\p{X_POSIX_Punct}`）可以匹配。这是因为Unicode把POSIX所认为的标点符号分为两类：标点（Punctuation）和符号（Symbols）。与`\p{punct}`不同，刚才提到的其他符号会匹配表5-14所示的字符。

表5-14：记为标点符号的ASCII符号

符号	码点	类别	文字系统	名
\$	U+0024	GC=Sc	SC=Common	DOLLAR SIGN
+	U+002B	GC=Sm	SC=Common	PLUS SIGN
<	U+003C	GC=Sm	SC=Common	LESS-THAN SIGN
=	U+003D	GC=Sm	SC=Common	EQUALS SIGN
>	U+003E	GC=Sm	SC=Common	GREATER-THAN SIGN
^	U+005E	GC=Sk	SC=Common	CIRCUMFLEX ACCENT
`	U+0060	GC=Sk	SC=Common	GRAVE ACCENT
	U+007C	GC=Sm	SC=Common	VERTICAL LINE
~	U+007E	GC=Sm	SC=Common	TILDE

还可以这样考虑：`[[:punct:]]`匹配Unicode认为是标点的所有字符（除非Unicode规则未生效），以及Unicode认为是符号的9个ASCII字符。

表5-15中的第二列给出了v5.14中可用的POSIX类。

表5-15: POSIX字符类

字符类	正常含义	有/a <sup>**</sup>
alnum	任何字母数字字符；也就是说，可以是任何alpha或digit。这包括很多非字母；见下一条。等价于\p{X_POSIX_Alnum}	仅匹配[A-Za-z0-9]。等价于\p{POSIX_Alnum} <sup>**</sup>
alpha	任何字母类字符，包括所有字母以及有Other_Alphabetic属性的非字母字符，如罗马数字、有圆圈的字母符号，以及有iota标志的希腊文字。等价于\p{X_POSIX_Alpha}	仅匹配52个ASCII字符 [A-Za-z]。等价于\p{POSIX_Alpha} <sup>**</sup>
ascii	序数值0~127的字符，等价于\p{ASCII}	匹配序数值为0~127的字符。等价于\p{ASCII} <sup>**</sup>
blank	任何水平空白符。等价于\p{X_POSIX_Blank}、\p{HorizSpace}或\h	仅匹配一个空格或一个制表符。等价于\p{POSIX_Blank}
cntrl	任何有Control属性的字符。通常这些字符不生成输出，而是以某种方式控制终端；例如，换行符、换页和退格都是控制字符。这个集合目前包括序数值0~31或127~159的所有字符。等价于\p{X_POSIX_Cntrl}	序数值0~31或127~159的所有字符。等价于\p{POSIX_Cntrl}
digit	任何有Digit属性的字符。更正式的说法是，有Numeric_Type=Decimal属性的字符，这包括数值递增（0到9）的连续10个字符，即Numeric_Value=0..9。等价于\p{X_POSIX_Digit} 或\d	10个字符：“0”到“9”。在/a下等价于\p{POSIX_Digit}或\d
graph	通用类别不是Control、Surrogate或Unassigned的所有非空白符字符。等价于\p{X_POSIX_Graph}	ASCII字符集中不包括空白符和控制字符的所有其他字符，所以为序数值33~126的字符。等价于\p{POSIX_Graph} <sup>**</sup>
lower	所有小写字符，不一定仅限于字母。包括通用类别 Lowercase_Letter的所有码点，以及有属性Other_Lowercase的字符。等价于\p{X_POSIX_Lower}或\p{Lowercase}。在/i下，也匹配GC=LC的字符，这是GC=Lu, GC=Lt和GC=Ll的缩写	只匹配26个ASCII小写字母 [a-z]。在/i下，还包括[A-Z]。等价于\p{POSIX_Lower} <sup>**</sup>



表5-15: POSIX字符类 (续)

字符类	正常含义	有/a <sup>*</sup>
print	任何graph或非控制blank字符。等价于 <code>\p{X_POSIX_Print}</code>	任何graph或非控制blank字符。等价于 <code>\p{X_POSIX_Print}</code> <sup>*</sup>
punct	通用类别为Punctuation的字符, 以及通用类别Symbol中的9个ASCII字符。等价于 <code>\p{X_POSIX_Punct}</code> 或 <code>\pP</code>	通用类别为Punctuation或Symbol的ASCII字符。等价于 <code>\p{POSIX_Punct}</code>
space	有Whitespace属性的字符, 包括制表符、换行、垂直制表符、换页、回车、空格、不可分空格、换行、窄空格、最小字间、Unicode段落分隔符以及大量空白符。等价于 <code>\p{X_POSIX_Space}</code> , <code>[\v\h]</code> , 或 <code>[\s\cK]</code> ; 如果只有 <code>\s</code> , 它等价于 <code>\p{X_Perl_Space}</code> , 这不包括 <code>\cK</code> , 即垂直制表符	所有ASCII空白符, 所以包括制表符、换行、垂直制表符、换页、回车和空格等价于 <code>\p{POSIX_Space}</code> ; 单独的 <code>\s</code> 表示不包括垂直制表符
upper	所有大写(但不是标题形式)字符, 但不一定仅限于字母。包括通用类别Uppercase_Letter的所有码点, 以及有Other_Uppercase属性的字符。如果有/i, 还包括与任何大写字母有相同大小写转换的字符。等价于 <code>\p{X_POSIX_Upper}</code> 或 <code>\p{Uppercase}</code>	只匹配26个大写ASCII字母[A-Z]。如果有/i, 还包括[a-z]。等价于 <code>\p{POSIX_Upper}</code> <sup>*</sup>
word	alnum字符, 或通用类别为Mark或Connector_Punctuation的字符。等价于 <code>\p{X_POSIX_Word}</code> 或 <code>\w</code> 。需要注意, Unicode标识符(包括Perl的标识符)要遵循它们自己的规则: 首字符有ID_Start属性, 后面的字符要有ID_Continue属性。(Perl还允许开头是Connector_Punctuation字符)	任何ASCII字母、数字或下划线。等价于 <code>\p{POSIX_Word}</code> , 或者在/a下等价于 <code>\w</code> <sup>*</sup>
xdigit	任何十六进制数字, 可以是窄ASCII字符, 或相应的宽字符。等价于 <code>\p{X_POSIX_XDigit}</code> , <code>\p{Hex_Digit}</code> 或 <code>\p{hex}</code>	ASCII范围内的所有十六进制数字。等价于 <code>[0-9A-Fa-f]</code> , <code>\p{POSIX_XDigit}</code> , <code>\p{ASCII_Hex_Digit}</code> 或 <code>\p{ahex}</code> 。

上表中标有<sup>\*</sup>的字符类在/a<sub>i</sub>下还能匹配某些非ASCII字符。目前这表示:

```

f U+017F GC=Ll SC=Latin LATIN SMALL LETTER LONG S
K U+212A GC=Lu SC=Latin KELVIN SIGN

```



因为第一个会大小写转换为正常的小写字母s，第二个会转换为正常的小写字母k。可以禁止这种大小写转换，将/a双写变成/aai。

在类名前加一个^前缀，后面跟[:，就可以将POSIX字符类取反（这是一个Perl扩展）。参见表5-16。

表5-16: POSIX字符类与其等价Perl字符类

POSIX	经典
[^:digit:]	\D
[^:space:]	\S
[^:word:]	\W

中括号是POSIX风格[::]构造的一部分，而不是整个字符类的一部分。这样一来，可以写类似/^[[:lower:]][[:digit:]]+\$/的模式，来匹配完全由小写字符或数字组成的字符串（以及可选的末尾换行符）。具体的，以下代码不能正常工作：

```
42 =~ /^[[:digit:]]$/          # 不正确
```

这是因为它不在一个字符类中。实际上，它本身是一个字符类，表示字符“:”、“i”、“t”、“g”和“d”。Perl并不关心你指定了两次“:”。你需要的代码应该是这样：

```
42 =~ /^[[:digit:]]+$/
```

POSIX字符类[.cc.]和[=cc=]能识别，不过会生成一个错误，指示不支持这些字符类。

## 量词

除非另作说明，否则正则表达式中的各项只匹配一次。对于类似/nop/的模式，每一个字符都必须匹配，而且必须按顺序依次匹配。像“panoply”或“xenophobia”等单词就可以匹配，因为匹配出现在哪里并不重要。

如果你想同时匹配“xenophobia”和“Snoopy”，就不能使用/nop/模式，因为这要求“n”和“p”之间只有一个“o”，而Snoopy中有两个“o”。这里就要用到量词了，量词指出了某个东西可以匹配多少次，而不是默认地只匹配一次。正则表达式中的量词就像是程序中的循环；实际上，如果把正则表达式看作是一个程序，那么它们确实是循环。有些循环要准确指定循环次数，如“这个匹配只重复5次”（{5}）。还有一些循环会同时指定匹配次数的上界和下界，如“这个匹配至少重复两次，但是不能超过4次”（{2,4}）。另外还有一些根本没有限定的上界，如“匹配至少两次，不过在此之上，想要多少次都可以”（{2,}）。

表5-17给出了Perl模式中可以识别的量词。

表5-17：正则表达式量词比较

最大	最小	占有	允许范围
<code>{MIN,MAX}</code>	<code>{MIN,MAX}?</code>	<code>{MIN,MAX}?+</code>	必须出现至少MIN次，但是不超过MAX次
<code>{MIN,}</code>	<code>{MIN,}?</code>	<code>{MIN,}?+</code>	必须出现至少MIN次
<code>{COUNT}</code>	<code>{COUNT}?</code>	<code>{COUNT}?+</code>	必须准确匹配COUNT次
<code>*</code>	<code>*?</code>	<code>*+</code>	0或多次（等同于 <code>{0,}</code> ）
<code>+</code>	<code>+</code>	<code>++</code>	1或多次（等同于 <code>{1,}</code> ）
<code>?</code>	<code>??</code>	<code>?+</code>	0或1次（等同于 <code>{0,1}</code> ）

如果有一个`*`或一个`?`，实际上并不需要匹配。这是因为，它可以匹配0次，而且仍认为是成功的。`+`通常可能是更好的选择，因为它要求必须匹配至少1次。

不要被上表中“准确”一词搞糊涂了。这只是指示重复次数，而不是整个字符串。例如，`$n =~ /\d{3}/`并不是说“这个字符串是3个数字长吗？”实际上它只是询问`$n`中有没有一个位置上连续出现3个数字。所以“101 Morris Street”之类的字符串测试为true，不过“95472”或“1-800-555-1212”之类的字符串也匹配。所有这些字符串都在某一处或多处包含3个数字，这正是你要求的。参见这一章后面“位置”一节，了解如何使用位置断言（如`/^\d{3}$/`）来确定位置。

如果某个部分匹配的次数可能不定，可以用最大量词将重复次数最大化。所以，如果说“想要多少次都可以”，贪婪量词把这解释为“匹配的次数尽可能多”，它只受一个需求的限制：不能导致后面指定的匹配失败。如果一个模式包含两个开放式量词，显然这两个量词都不能消费整个字符串：匹配中一部分使用的字符对于另一部分来说不再可用。每个量词都是贪婪的，而跟在它后面的量词就要做出“牺牲”，它们会从左向右读取模式。

这是正则表达式中量词的传统行为。不过，Perl允许改变量词的行为：通过在量词后面放置一个`?`，可以把它从最大量词转换为最小量词。这并不是说最小量词总是要匹配其范围允许的最小重复次数，这不像最大量词那样必须匹配其范围允许的最大次数。首先整个匹配仍必须成功，最小匹配会为了成功尽可能多地取所需要的字符，但不能过多（不能影响匹配成功）。最小量词更强调满意而不是贪婪。

例如，在以下匹配中：

```
"exasperate" =~ /e(.*)e/      # $1现在是"xasperat"
```

`.*`匹配“xasperat”，这是它能匹配的最长的字符串（它还将这个值存储在`$1`中，参见这一章后面“分组和捕获”一节的介绍）。尽管还可以有一个更短的匹配，不过贪婪匹配不关心这个短匹配。如果在相同的起点上有两个选择，它总是返回二者中较长的一个。

与下面的匹配做个对比：



```
"exasperate" =~ /e(.*)e/    # $1现在是"xasp"
```

在这里，使用了最小匹配`.*?`。为`*`增加一个`?`，这就构成一个`*?`，它会有相反的行为：现在如果在相同的起点上有两个选择，它总是返回二者中较短的一个。

尽管可以把`*?`读作匹配某个东西0次或多次，不过谈到0次，这并不是说它总能匹配0个字符。例如，如果这里是这样做的，从而将`$1`设置为`"`，那么将无法找到第二个“e”，因为它并没有直接跟在第一个“e”的后面。

你可能还会奇怪：为什么在最小匹配`/e(.*)e/`中Perl没有把“rat”放在`$1`中？毕竟，“rat”也落在两个e之间，而且它比“xasp”更短。在Perl中，只有在多个有相同起点的匹配中选择最短或最长匹配时才会应用最小/最大选择。如果存在两个可能的匹配，不过它们在字符串中有不同的起点，那么它们的长度并不重要，另外使用一个最小量词还是一个最大量词也不重要。在这些合法的匹配中，最早出现的匹配往往会胜出，超越所有后来者。只有当多个可能的匹配都从相同的起点开始时，才能用最小或最大匹配来打破僵局。如果起点不同，就没有僵局需要打破。正常情况下Perl的匹配是“最左最长匹配”（leftmost longest）；对于最小匹配则变成“最左最短匹配”（leftmost shortest）。不过“最左”部分没有变，这也是基本原则<sup>注13</sup>。

有两种方法可以消除模式匹配引擎的左倾向性。首先，可以使用一个先出现的贪婪量词（一般是`*`）消费掉（吞掉）字符串中前面的部分。搜索一个贪婪量词的匹配时，它会首先尝试最长的匹配，实际上这会从右向左搜索字符串的其余部分：

```
"exasperate" =~ /. *e(.*)e/    # $1现在是"rat"
```

不过这样做时要当心，因为现在整个匹配会包含直到这一点的整个字符串。

第二种解除左倾向性的方法是使用位置断言，这将在下一节讨论。

只需在量词后面增加一个`?`，可以将任何最大量词改变为最小量词，同样的，也可以在量词后面增加一个`+`，将任何最大量词改为一个占有量词。占有匹配是一种控制回溯的方法。最小和最大量词总是尝试所有可能的组合来查找一个匹配，而占有量词不会回溯来尝试查找另一个可能性，这会大大提高性能。

对于简单匹配来说，这通常不是问题，不过一旦有多个量词，尤其是多个嵌套的量词，这就很有影响了。通常整个匹配是否成功并不会改变，不过如果匹配会失败，则利用占有量词可以更快地失败。例如：

---

注13：并不是所有正则表达式引擎都是这样工作的。有些引擎坚信整体贪婪匹配，在这种情况下，总是最长的匹配胜出，尽管它有可能后出现。Perl没有采用这种方式。你可能称之为“热情”胜过“贪婪”（或吝啬）。关于这个原则以及很多其他原则更正式的讨论，请参见“小引擎/能与（不能）/”一节。



```
("a" x 20 . "b") =~ /(a*a*a*a*a*a*a*a*a*a*a*a*a*a)*[^\Bb]$/
```

最后这个匹配终将失败。正则表达式引擎会很努力地工作，徒劳地尝试所有可能的a和\*的组合。它没有意识到最终将会失败。通过把一个或多个可变的\*量词改为一个不可变的\*+量词，可以让它更快地失败。在这里，通过改变最后一个量词，从最大量词转变为占有量词，可以使性能有两个量级的提升，这可不容小视。

当然，这是一个人为设想的极端例子，不过构造复杂的模式时，很可能出现这种问题，而你有可能还没有意识到。事实上，占有量词的工作与我们后面将要遇到的非回溯组很类似。占有匹配`a*+`与`(?>a*)`相同。占有组比占有量词稍稍灵活一些，因为可以把多个东西分组在一起构成一个单元，它对回溯是不可见的。不过占有量词输入要容易得多，而且需要避免灾难性的回溯时，使用占有量词就足够了。

## 位置

有些正则表达式构造会表示字符串中要匹配的位置 (position)，这是指一个实际字符左边或右边的位置。这些元符号就是0宽度断言的例子，因为它们不对应字符串中具体的字符。我们通常把它们称为“断言”（它们也称为“锚”，因为可以将模式的某个部分锚定在一个特定的位置，而模式的其余部分可以“随波逐流”）。

不必使用模式也可以处理字符串中的位置。利用内置的`substr`函数，你可以抽取子串并赋值，可以从字符串开始算起，或者从字符串末尾算起，或者从一个特定的数值偏移开始算。如果你要处理定长的记录，这可能就是你想要的全部。只有当数值偏移还不够时才有必要使用模式。不过，大多数情况下，`substr`都是不够的，或者至少与模式相比还不够方便。

## 开头：\A和^断言

不论字符串中是什么，`\A`断言只在字符串的开头匹配。不过，相比之下，`^`断言不仅匹配字符串的开头，它还能匹配更为传统的行开头：如果模式使用`/m`修饰符，而且字符串中有嵌套的换行符，`^`还能匹配字符串中直接跟在一个换行符后面的任何内容：

```

/\Abar/      # 匹配"bar"和"barstool"
/^bar/      # 匹配"bar"和"barstool"
/^bar/m     # 匹配"bar"、"barstool"和"sand\nbar"

```

与/g结合使用时，/m修饰符允许^在同一个字符串中匹配多次：

```
s/^s+//gm;           # 去除每一行的前导空白符
$total++ while /^./mg; # 统计非空行数
```

## 结尾：\z、\Z和\$断言

不论字符串中是什么，\z元符号只在字符串结尾匹配。如果有换行符，\Z还会匹配字符串末尾紧挨在换行符前面的内容，或者如果没有换行符，则匹配字符串末尾的内容。\$元字符通常与\Z含义相同。不过，如果指定了/m修饰符，而且字符串中有嵌套的换行符，\$还能在字符串中紧挨在换行符前面的位置匹配：

```
/bot\z/      # 匹配 "robot"
/bot\Z/      # 匹配 "robot" 和 "abbot\n"
/bot$/      # 匹配 "robot" 和 "abbot\n"
/bot$/m     # 匹配 "robot"、"abbot\n"和"robot\nrules"
/^robot$/   # 匹配 "robot" 和 "robot\n"
/^robot$/m  # 匹配 "robot"、"robot\n" 和 "this\nrobot\n"
/\Arobot\Z/ # 匹配 "robot" 和 "robot\n"
/\Arobot\z/ # 只匹配 "robot" — 不过为什么不用eq呢？
```

与^类似，与/g结合使用时，/m修饰符允许\$在同一个字符串中匹配多次（这些例子假设你已经将一个多行记录读入\$\_，可能在读取之前将\$/设置为"")。

```
s/\s*$/gm; # 去除段落中各行的末尾空白符

while (/^([^:]+):\s*(.*)/gm ) { # 得到邮件首部
    $headers{$1} = $2;
}
```

在这一章后面的“变量内插”一节中，我们将讨论如何将变量内插到模式中：如果\$foo是“bc”，则/a\$foo/等价于/abc/。在这里，\$不匹配字符串的结尾。要让\$匹配字符串的结尾，它必须位于模式的末尾，或者后面紧跟着一个竖线或结束小括号。

## 边界：\b和\B断言

\b断言匹配任何单词边界，单词边界定义为\w字符和\W字符之间的一个位置，\w字符和\W字符的顺序可以任意。如果顺序是\W\b，则是一个单词开始边界，如果顺序是\b\W，则是单词结束边界（这里字符串结尾作为\W字符）。\B断言匹配所有不是单词边界的位置，也就是说，可以匹配\w\w或\W\W的中间位置。

```
/\bis\b/ # 匹配 "what it is" 和 "that is it"
/\Bis\B/ # 匹配 "thistle" 和 "artist"
/\bis\B/ # 匹配 "istanbul" 和 "so— isn't that butter?"
/\Bis\b/ # 匹配 "confutatis" 和 "metropolis near you"
```

由于\W包括所有标点符号字符（除了下划线），所以在字符串中间可以有\b边界，如“isn't”，“booktech@oreilly.com”，“M.I.T.”和“key/value”。

在中括号字符类中([\b])，\b表示一个退格而不是单词边界。

## 渐进匹配

结合使用/g修饰符时，pos函数允许你读取或设置下一个渐进匹配的开始偏移量：

```
$burglar = "Bilbo Baggins";
while ($burglar =~ /b/gi) {
    printf "Found a B at %d\n", pos($burglar)-1;
}
```

我们将位置减1，这是因为这才是我们想要查找的字符串的长度，pos往往是成功匹配之后的那个位置。

以上代码会打印以下结果：

```
Found a B at 0
Found a B at 3
Found a B at 6
```

如果匹配失败，匹配位置通常会重置为起始位置。如果还应用了/c修饰符（表示“继续”），/g处理完所有字符后，尽管匹配失败，但不会重置位置指针。这允许你在这个位置之后继续你的搜索，而不用从开头重新开始：

```
$burglar = "Bilbo Baggins";
while ($burglar =~ /b/gci) { # ADD /c
    printf "Found a B at %d\n", pos($burglar)-1;
}
while ($burglar =~ /i/gi) {
    printf "Found an I at %d\n", pos($burglar)-1;
}
```

除了之前找到的3个B，Perl现在还报告在位置10找到一个i。如果没有/c，则第二个循环的匹配会从开头重新开始，这会先在位置1找到另一个i。

## 停止：\G断言

只要开始从pos函数的角度来考虑，就很容易想到用substr划分字符串，不过这通常并不合适。更常见的，如果开始模式匹配，就应该继续使用模式匹配。不过，如果你想找一个位置断言，则可能要找的是\G。

\G断言在模式中表示一个位置，这与pos在模式外表示的位置含义相同。用/g修饰符渐进地匹配一个字符串时（或者使用pos函数直接选择起点），可以使用\G来指定上一次匹配后面的那个位置。也就是说，它会匹配紧挨着pos所标识的字符之前的位置。这样一来，你就能记住是从哪里离开的：

```
($recipe = <<'DISH') =~ s/^\s+//gm;
    Preheat oven to 451 deg. Fahrenheit.
    Mix 1 ml. dilithium with 3 oz. NaCl and
    stir in 4 anchovies. Glaze with 1 g.
```



```
mercury. Heat for 4 hours and let cool
for 3 seconds. Serves 10 aliens.
DISH
```

```
$recipe =~ /\d+ /g;
$recipe =~ /\G(\w+)/;      # $1现在是"deg"
$recipe =~ /\d+ /g;
$recipe =~ /\G(\w+)/;      # $1现在是"ml"
$recipe =~ /\d+ /g;
$recipe =~ /\G(\w+)/;      # $1现在是"oz"
```

\G元符号通常在循环中使用，如下一个例子所示。每个数字序列之后我们会“暂停”，在这个位置上，我们将测试是否有一个缩写。如果有，就获取接下来两个单词。否则，只获取下面一个单词：

```
pos($recipe) = 0;      # 只是为了安全，将\G重置为0
while ( $recipe =~ /(\d+) /g ) {
    my $amount = $1;
    if ($recipe =~ /\G (\w{0,3}) \. \s+ (\w+) /x) {      # 缩写. + 单词
        say "$amount $1 of $2";
    } else {
        $recipe =~ /\G (\w+) /x;      # 就是一个单词
        say "$amount $1";
    }
}
```

这会生成以下结果：

```
451 deg of Fahrenheit
1 ml of dilithium
3 oz of NaCl
4 anchovies
1 g of mercury
4 hours
3 seconds
10 aliens
```

## 分组与捕获

可以将模式的不同部分分组为子模式，并记住与这些子模式匹配的字符串。我们把第一个行为称为分组（grouping），第二个行为称为捕获（capturing）。也可以只分组而不捕获。稍后会介绍更多有关内容。

### 捕获

要捕获一个子串以备以后使用，可以用小括号包围与它匹配的子模式。第一对小括号将其子串存储在\$1中，第二对小括号将子串存储在\$2，依此类推。你可以根据需要使用任意多个小括号；Perl会继续为你定义更多编号变量，来表示捕获的这些字符串。

下面来看几个例子：

```
/(\d)(\d)/    # 匹配两个数字，将它们捕获到$1和$2中
/(\d+)/       # 匹配一个或多个数字，将它们都捕获到$1中
/(\d)+/       # 匹配一个数字一次或多次出现，将最后一个捕获到$1中
```

注意第二个和第三个模式之间的差别。通常你想要的是第二种形式。第三种形式不会为多位数字创建多个变量。会在编译模式时为小括号编号，而不是在匹配时编号。

捕获的字符串通常称为组引用（group references），因为它们指回所捕获的部分文本。原先的模式匹配引擎限制组引用只能是反向引用（backreferences），不过Perl允许引用任何组，可以是反向、正向或者甚至可以是中间。

要得到这些捕获组，实际上有两种方法。你已经看到了编号变量，可以利用这些编号变量在模式之外得到反向引用，不过这种方法在模式中不适用。必须使用反向引用记法，所以应当使用\1，\2，\g{1}，\g{2}，\k<some\_group>，\k<other\_group>等。

在模式中，不能使用\$1作为组引用，因为编译正则表达式时这已经作为一个普通变量内插到字符串中。所以我们要在模式中使用传统的\1组引用记法。对于两位和三位反向引用数字，则存在二义性，可能会与八进制字符记法混淆，不过这个问题可以很好地解决，只需要考虑有多少可用的捕获模式。例如，如果Perl看到一个\11元符号，只有当模式中之前至少捕获了11个子串时，这才等价于\$11。否则，这就等价于\011，也就是一个制表符。为了避免这种二义性，应当使用\g{NUMBER}按编号指示捕获组，而使用\o{OCTNUM}按编号指示八进制字符。所以\g{11}总是第11个捕获组，而\o{11}总是码点为八进制11的字符。不过与罗列11个捕获组的做法相比，更好的想法是使用命名组，见下面的介绍。

所以，要查找双字如“the the”或“had had”，可以使用以下模式：

```
/\b(\w+) \1\b/i
```

不过，最常见的还是使用\$1形式，因为通常都会应用一个模式，然后用子串做些处理。假设有类似下面的一些文本（这是一个邮件首部）：

```
From: gnat@perl.com
To: camelot@oreilly.com
Date: Mon, 17 Jul 2011 09:00:00 -1000
Subject: Eye of the needle
```

你希望构造一个散列，将每个冒号前的文本映射到冒号后的文本。如果逐行循环处理这些文本（因为你在从一个文件读取这些文本），可以这样做：

```
while (<>) {
    /^(.*?): (.*)$/; # 冒号前的文本捕获到$1，冒号后的文本捕获到$2
    $fields{$1} = $2;
}
```

与\$`、\$&和\$'类似，这些编号变量是动态作用域变量，其作用域延伸到外围块或eval字符串的末尾，或者延伸到下一个成功的模式匹配，这要看哪一个先出现。还可以把它们用在模式替换的右边（替换部分）：

```
s/^(\\S+) (\\S+)/$2 $1/;      # 交换前两个单词
```

分组可以嵌套，如果有嵌套，这些分组要按左括号的位置计数。所以如果有字符串“Primula Brandybuck”，以下模式：

```
/^((\\w+) (\\w+))$/
```

将把“Primula Brandybuck”捕获到\$1中，“Primula”捕获到\$2中，“Brandybuck”捕获到\$3中，如图5-1所示。

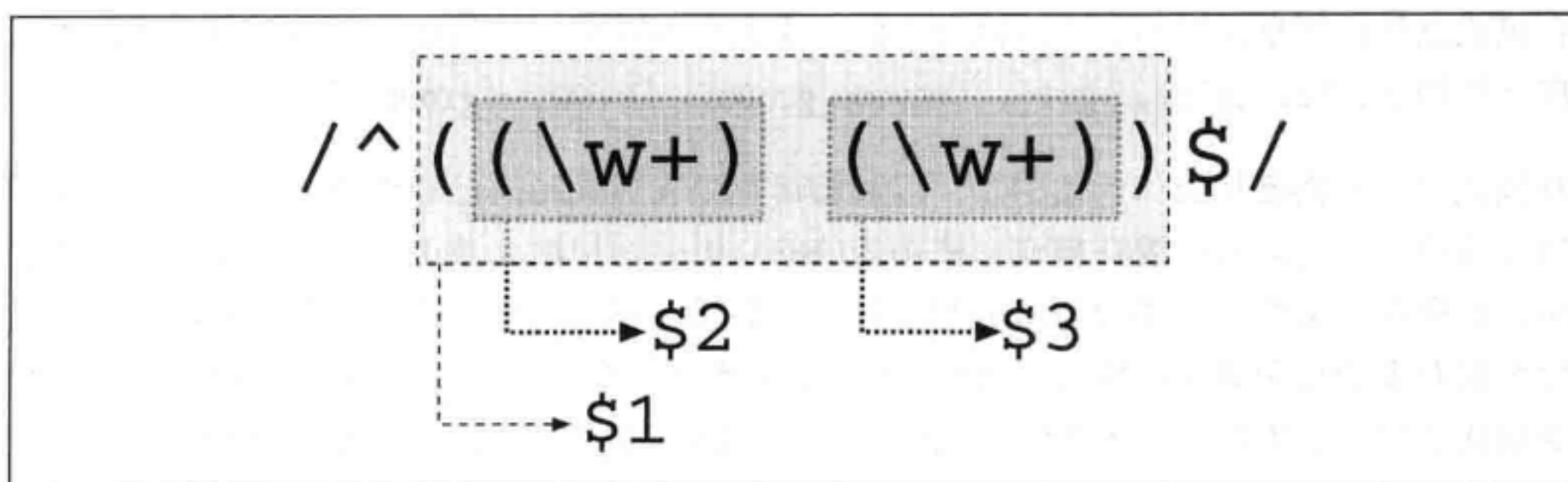


图5-1：用小括号创建组引用

前面提到过，并不是所有组引用都一定是反向引用。你可以引用模式匹配的任何组，即使组中还没有填充具体的子串。只有当你要做某种重复处理，会反复访问同一个组时，这才有用。第一次遇到这些非反向引用的组引用时，它们会失败，因为这些匹配还没有发生。不过下一次访问时，这个组就有些意思了。

下面是捕获组的3类引用。第一类是传统的反向引用，因为在首次请求时它已经完成：

```
"foofoobar" =~ /^(foo)\1bar$/      # 反向引用
```

不过，下面是一个前向引用：

```
"foofoobar" =~ /^((\\3bar)|(foo))+$/  # 前向引用
```

第一次处理+量词的重复时，甚至还没有开始第3个组，所以\\3会失败，我们跳至其他候选项，这会用字符串的第一个“foo”填充第3个组。下一次再处理+量词时，\\3包含“foo”，所以最后得到bar，处理结束。

第3个例子既不是反向引用，也不是一个前向引用，因为它在所引用的组中间，这使它成为一种环向引用（circumref）：



```
"foofoobar" =~ /^(\1bar|(foo))+/      # 环向引用
```

带捕获的模式通常用于列表上下文，用来填充一个值列表，因为模式很聪明，会把捕获的子串作为一个列表返回：

```
($first, $last) = /^(\w+) (\w+)$/;
($full, $first, $last) = /^((\w+) (\w+))$/;
```

利用/g修饰符，模式可以从多个匹配返回多个子串，所有子串都放在一个列表中。假设将之前看到的邮件首部放在一个字符串中（例如，放在\$\_中）。可以像逐行循环一样做同样的处理，不过只需要一条语句就足够了：

```
%fields = /^(.*?): (.*$)/gm;
```

这个模式匹配4次，每次匹配时，它会找到两个子串。/gm匹配将所有这些子串作为一个扁平列表返回，其中包含8个字符串，对%fields完成列表赋值可以很方便地把这个列表解释为4个键/值对，大功告成。

还有其他一些特殊变量可以处理模式匹配中捕获的文本。\$&包含整个匹配的字符串，\$`包含匹配左边的字符串，\$'包含匹配右边的字符串。\$+包含最后一个捕获组的内容。

```
$_ = "Speak, <EM>friend</EM>, and enter.";
m[ (<.*?>) (.*?) (</.*?>) ]x; # 标记，然后是字符，然后是一个结束标记
say "prematch: $`";           # Speak,
say "match: $&";              # <EM>friend</EM>
say "postmatch: $'";          # , and enter.
say "lastmatch: $+";          # </EM>
```

关于这些神奇的精灵变量（以及如何用英文书写），更多的解释请参见第25章。

@- (@LAST\_MATCH\_START)数组包含子匹配开头的偏移量，@+ (@LAST\_MATCH\_END)包含子匹配结尾的偏移量：

```
#!/usr/bin/perl
$alphabet = "abcdefghijklmnopqrstuvwxyz";
$alphabet =~ /(hi).*(stu)/;
say "The entire match began at $-[0] and ended at $+[0]";
say "The first match began at $-[1] and ended at $+[1]";
say "The second match began at $-[2] and ended at $+[2]";
```

如果确实想匹配一个直接量小括号字符，而不是把它解释为一个元字符，需要加一个反斜线：

```
/\ (e.g., .*?) \/
```

这会匹配一个有括号的例子，如(e.g., this statement)，这个语句就能匹配。不过由于点号是一个通配符，它也会匹配任何第一个字母为e而且第三个字母为g的有括号的语句，如(ergo, this statement, too)，这个语句也能匹配。

编号捕获组本质上很脆弱。假设使用类似这样的模式来匹配一个重复单词序列：

```
$dupword = qr/ \b (?: ( \w+ ) (?: \s+ \1 )+ ) \b /xi;
```

如果把它嵌在一个更大的模式中，这个模式本身也有捕获组，而且出现在你的重复单词模式前面，那么\1就是错误的。例如，这样是不行的：

```
$quoted = qr{ ( [''] ) $dupword \1 }x;
```

这是因为，如果\$dupword嵌套在\$quoted中，在这里\$dupword就应该使用\2。不过你不能这么做，因为\$dupword编译时，还没有第二个捕获组可以引用，所以它甚至不能编译。

对于这种情况，一种解决方案是使用相对编号捕获组。要访问这些捕获组，需要使用\g{NUMBER}记法，这是一个编号引用，指向捕获组NUMBER。NUMBER为正数时，这与\NUMBER相同。不过NUMBER为负数时，则表示之前多少个捕获组。所以\g{-1}就是最后一个捕获组，\g{-2}是倒数第二个捕获组，依此类推。

对于重复单词模式，一个更好的定义如下，采用这种方式定义时，它能更自由地嵌套在一个更大的模式中：

```
$dupword = qr/ \b (?: ( \w+ ) (?: \s+ \g{-1} )+ ) \b /xi;
```

下面是一个简单的程序，可以在输入流中逐段地查找重复单词序列：

```
#!/usr/bin/env perl
use v5.14;

my $dupword = qr/ \b (?: ( \w+ ) (?: \s+ \g{-1} )+ ) \b /xi;
my $quoted = qr{ ( [''] ) $dupword \1 }x;
$/ = q(); # 跨段落

while (<>) {
    while (/ $quoted /pg) {
        printf "%s %d: %s\n", $ARGV, $., ${^MATCH};
    }
} continue {
    close ARGV if eof;
}
```

尽管这个程序本身能很好地工作，但还存在一个严重的问题。如果\$quoted用在一个更大的模式中，使用\1来引用它就是错误的。不过它无法知道倒数多少个捕获组，因为它不必知道\$dupword中无关的模式使用了多少个捕获组。

## 命名捕获组

对于最后这个难题，唯一的解决办法就是需要一个新策略，而不使用编号捕获组。为此（以及很多其他原因），我们发明了命名捕获组（named capture groups）。要在你的模式中声明一个命名捕获组，可以使用(?<NAME>...)。这仍是一个捕获组，就像常规的加括号

的分组一样，不过它的名字是`NAME`。

或者，更确切地说，它的名字也是`NAME`，因为命名捕获组同时也是一个编号捕获组，就好像这个编号捕获组在相同的位置上，只是没有名字。这与Java和Python最常用的正则表达式库中命名组与编号组的做法是类似的。不过，这与.NET Framework（如C#）中命名组的做法不同，在.NET Framework中，在所有编号组之后才会为命名捕获组指定编号（在另一个奇怪的语言Perl 6中，只有当一个捕获没有其他名字时才会为它指定一个编号。你可以试试看）。

要反向引用相同模式中的一个命名捕获组，类似于对编号组使用`\1`或`\g{1}`，可以使用`\k<NAME>`。对于前一个例子中有问题的`$quoted`定义，现在我们可以利用命名捕获组来解决，从而允许它用在更大的模式中：

```
$quoted = qr{ (?<quote> ['"] ) $dupword \k<quote> }x;
```

把它放在模式中是可以的，不过模式匹配中由于组`\1`可以作为`$1`访问，有时你可能希望在模式运行之后还能访问命名捕获组的内容。这正是内置散列变量`%+`的作用。散列中的键就是你为捕获组指定的名字，值就是这些组捕获的内容。所以，根据前面给出的`$quoted`和`$dupword`的定义，可以如下取出引用的所有重复单词序列：

```
say ${quote} while /$quoted/g;
```

下面再给出一个例子：

```
$word = "bookkeeper";  
$word =~ s/ (?<letter> \p{alpha} ) \k<letter> /${letter}/gix;  
# $word 现在是"bokeper"
```

这里我们使用了`/x`修饰符，这只是为了能够在模式中放入一些空白符，以方便阅读。没人愿意读类似“Nobodylikesreadingthiskindofthing”的文字。

如果同一个模式中有多个组同名，都名为`NAME`，`%+`只包含捕获的最后一个字符串，不过`%-`散列中的各项会保存一个数组引用，相应数组中包含这些同名捕获组捕获的所有内容。如果有多个名为`NAME`的捕获组，使用`@{${NAME}}`可以得到所有这些捕获组，使用`${NAME}[0]`可以得到第一个，用`${NAME}[1]`可以得到第二个，依此类推，直到`${NAME}[-1]`得到最后一个捕获组。

假设你想先匹配名然后匹配编号，或者先匹配编号再匹配名。如果使用编号捕获组，就无法知道哪个是哪个：

```
/ (\d+) \s+ (\pL+) | (\pL+) \s+ (\d+) /x
```

因为现在你不知道走的是哪一个分支，所以也不知道获取`$1`和`$2`还是获取`$3`和`$4`。这里就要用到`(?|...)`分支重置构造：



```

m{
    (?| (\d+) \s+ (\pL+)      # 这些是$1和$2
      | (\pL+) \s+ (\d+)      # 这两个也是!
    )
}x

```

对于每一个候选项，组编号都会重置为到进入这个分支重置构造时的编号。这样一来，不论哪一个候选项匹配，你都知道数据在\$1和\$2中。

这对于比较简单的模式是可以的，不过从长远看，另外对于更复杂的模式来说，使用命名捕获组会更好。通过像这样为它们指定名字：

```

m{
    (?<name> \pL+ ) \s+ (?<number> \d+ )
    |
    (?<number> \d+ ) \s+ (?<name> \pL+ )
}x

```

现在走哪个选择分支并不重要。匹配之后，可以这样访问匹配组的内容：

```

${name}
${number}

```

不过，如果没有候选项，两个捕获会加载两次：

```

m{
    (?<name> \pL+ ) \s+ (?<number> \d+ )
    \W+
    (?<number> \d+ ) \s+ (?<name> \pL+ )
}x

```

所以，如果模式匹配，<name>和<number>组分别会有两个值。同一个命名组加载超过一次时，%+变量中之前的内容会被覆盖，这就像对一个标量变量赋值多次。不过，%-变量会在散列中对应各个名保存相应的值数组，所以新内容会压入到与命名键关联的匿名数组的末尾。

%+中的值是字符串，而%-中的值是数组引用，所以可以像这样得到整个匹配集：

```

@{ ${name} }
@{ ${number} }

```

或者像这样得到单个标量：

```

${name}[0]
${name}[1]
${number}[0]
${number}[1]

```

这样一来，\${name}就等同于\${name}[-1]，另外与\${name}[\$#{name}]也相同，不过这样有些烦琐。

顺便说一句，如果你不希望用单个标点符号作为变量名，标准Tie::Hash::NamedCapture模

块允许你对这两个内置散列使用你喜欢的任何名字。如果你喜欢类似%-变量的形式，可以传入一个额外的参数`all => 1`；否则，这个绑定的散列就类似于%+变量。

```
use Tie::Hash::NamedCapture;
tie my %last_captured, "Tie::Hash::NamedCapture";
tie my %all_captured, "Tie::Hash::NamedCapture", all => 1;
```

可以通过这些变量访问你的命名捕获组，就像使用\$+和%+一样，不过现在可以用你自己的名字来访问：

```
$last_captured{name}
$last_captured{number}

@{ $all_captured{name} }
@{ $all_captured{number} }

$all_captured{name}[0]
$all_captured{name}[1]

$all_captured{number}[0]
$all_captured{number}[1]
```

从命名捕获组的使用应该能看到，除了最简单的模式外，最好使用命名捕获组而不是编号捕获组（有人说，即使是最简单的模式也应该使用命名捕获组）。不过当你写递归模式和文法时，命名捕获组才会真正展现出它的巨大魅力，有关描述参见后面的“非传统模式”一节。

## 分组而不捕获

裸小括号可以完成分组和捕获。不过有时你并不希望这样。有些情况下你只想对模式的某部分分组，而不需要捕获字符串以备以后使用。小括号有一种扩展形式，即`(?:PATTERN)`记法，可以达到这个目的。

之所以希望分组而不捕获，至少有3个原因：

1. 要对某个东西量化。
2. 要限制内部选项的作用域。例如`/^cat|cow|dog$/`应当是`/^(?:cat|cow|dog)$/`，这样`cat`就不会随`^`一起跑掉。
3. 将一个嵌套模式修饰符的作用域限制为一个特定的子模式，如`/foo(?-i:Case_Matters)bar/i`中（参见下一节“作用域模式修饰符”）。

另外，对于你不打算使用的东西，禁止捕获这些部分还能提高效率。从负面来看，这种记法看起来有点混乱。

在一个模式中，如果左小括号后面紧跟着一个问号，这会指示一个正则表达式扩展。目前

的正则表达式集相对固定，我们不太敢创造新的元字符，因为害怕新元字符会破坏原来的Perl程序。我们更愿意使用这种扩展语法为正则表达式集增加新特性。

在本章后面，我们会看到更多的正则表达式扩展，所有这些都只是分组而没有捕获，另外还会完成其他工作。(?:*PATTERN*)扩展的特殊性在于它不做其他任何事情。所以如果有：

```
@fields = split(/\b(?:a|b|c)\b/)
```

这就像：

```
@fields = split(/\b(a|b|c)\b/)
```

不过，不会划分的额外的字段（`split`操作符有点像`m//g`，它会为模式中所有捕获子串划分出额外字段。正常情况下，`split`只返回不匹配的部分。有关`split`的更多内容请参见第27章）。

## 作用域模式修饰符

可以将`/i`、`/m`、`/s`、`/x`、`/d`、`/u`、`/a`、`/l`和`/p`修饰符的词法作用域限定在模式中的某一部分，需要把它们插入到（不加斜线）分组记法的`?`和`:`之间。如果有：

```
/Harry(?:i:s) Truman/
```

这会匹配“Harry S Truman”和“Harry s Truman”，而

```
/Harry(?:x: [A-Z] \.? \s )?Truman/
```

会匹配“Harry S Truman”和“Harry S. Truman”，以及“Harry Truman”；另外：

```
/Harry(?:ix: [A-Z] \.? \s )?Truman/
```

能匹配所有这5个字符串，它在组中结合了`/i`和`/x`。

还可以用一个负号从作用域中减去修饰符：

```
/Harry(?:x-i: [A-Z] \.? \s )?Truman/i
```

这会匹配这个名字的任何大写，不过如果提供了中间名缩写，它必须是大写的，因为应用到整个模式的`/i`在这个作用域中会被取消。

要减去修饰符时，只能减去`/i`、`/m`、`/s`或`/x`。其他修饰符`/d`、`/u`、`/a`、`/l`和`/p`只能增加。

通过省略冒号和`PATTERN`，可以把修饰符设置导出到一个外部组，将它变成一个作用域。也就是说，可以为该修饰符小括号外面一层的分组构造有选择地打开和关闭修饰符，如下：

```
/(?i)foo/           # 等价于/foo/i
```



```
/foo(?:-i)bar)/i      # "bar" 必须是小写
/foo(?:x-i) bar)/      # 启用/x，并对"bar"禁用/i
```

需要说明，第二个和第三个例子会创建捕获组。如果这不是你想要的，则应该分别使用 `(?-i:bar)` 和 `(?x-i: bar)`。如果你希望“.”匹配模式中某一部分中的换行符，但不匹配其余部分中的换行符，这种情况下，对模式中的一部分设置修饰符尤其有用，而对整个模式设置 `/s` 没有任何帮助（除非你使用 `\N` 来匹配非换行符）。

## 候选项

在一个模式或子模式中，使用 `|` 元字符可以指定一组可能性，其中任何一个可能都可以匹配。例如：

```
/Gandalf|Saruman|Radagast/
```

可以匹配 `Gandalf` 或 `Saruman` 或 `Radagast`。候选只能延伸到最内层外围小括号（不论是否捕获）：

```
/prob|n|r|l|ate/      # 匹配 prob、n、r、l 或 ate
/pro(b|n|r|l)ate/      # 匹配 probate、pronate、prorate 或 prolate
/pro(?:b|n|r|l)ate/    # 匹配 probate、pronate、prorate 或 prolate
```

第二种和第三种形式会匹配相同的字符串，不过第二种形式将把可变字符捕获到 `$1` 中，而第三种形式不会捕获。

在任何给定位置，引擎总会尝试匹配第一个候选项，然后是第二个，依此类推。候选项的相对长度并不重要，这说明，在以下模式中：

```
/(Sam|Samwise)/
```

不论与哪个字符串匹配，`$1` 绝对不会设置为 `Samwise`，因为 `Sam` 总会先匹配。如果有类似这样的重叠匹配，要把较长的字符串放在前面。

不过候选项的顺序只在给定的位置上有影响。引擎的外层循环会完成从左到右的匹配，所以下面的代码总会匹配第一个 `Sam`：

```
"Sam I am," said Samwise" =~ /(Samwise|Sam)/; # $1 eq "Sam"
```

要强制为从右到左扫描，需要使用贪婪量词：

```
"Sam I am," said Samwise" =~ /(Samwise|Sam)$/; # $1 eq "Samwise"
```

可以包含我们之前了解的某个位置断言来消除从左到右（或从右到左）匹配，如 `\G`、`^` 和 `$`。这里我们将模式锚定到字符串的末尾：

```
"Sam I am," said Samwise" =~ /(Samwise|Sam)$/; # $1 eq "Samwise"
```

注意这里将\$放在候选项的外面（因为我们已经加了一对小括号，所以可以把它放在后面），不过如果没有这样的小括号，还可以把断言分配给任何一个候选项或者分配给所有候选项，这取决于你希望它们如何匹配。下面这个小程序会显示以一个\_\_DATA\_\_或\_\_END\_\_ token开头的行：

```
#!/usr/bin/perl
while (<>) {
    print if /^__DATA__|^__END__;/;
}
```

不过这样做时要当心。要记住，第一个和最后一个候选项（第一个|前面的候选项和最后一个候选项）很可能会“吞掉”两边正则表达式中的其他元素，直到表达式结束，除非有外围小括号。以下就是一个常见的错误：

```
/^cat|dog|cow$/
```

实际上我们想要的是：

```
/^(cat|dog|cow)$/
```

第一个模式会匹配开头是“cat”的字符串，或任何位置上有“dog”的字符串，再或者是末尾有“cow”的字符串。第二个模式会匹配仅由“cat”或“dog”或“cow”组成的字符串。它还会捕获\$1，这可能不是你想要的。可以用下面的模式来限制：

```
/^cat$|^dog$|^cow$/
/^(?:cat|dog|cow)$/
```

一个候选项可以为空，如果某个候选项为空，那么它总能匹配：

```
/com(pound|)/;      # 匹配"compound" 或 "com"
/com(pound(s|)|)/;  # 匹配"compounds", "compound"或 "com"
```

这非常类似于使用?量词，它会匹配0次或1次：

```
/com(pound)?/;      # 匹配"compound" 或 "com"
/com(pound(s?))?/;   # 匹配"compounds", "compound"或"com"
/com(pounds?)/;     # 与上面一样，但是没有使用$2
```

不过这里还有一个区别。将?应用到一个将捕获到编号变量中的子模式时，如果其中没有字符串，这个变量将是未定义的。如果使用一个空候选项，尽管仍为false，但变量是一个已定义的空串。

## 保持控制

好的经理都知道，不要对员工施加微观控制。只要告诉他们你想要什么，让他们自己去寻找解决问题的最佳方法。类似的，通常最好把正则表达式看作是某种规范：“这是我想要的，去找出满足要求的一个字符串。”

另一方面，最优秀的经理还会了解员工要完成的任务。Perl中的模式匹配也是如此。对于Perl将如何匹配某个特定的模式，你了解得越深入，就能越聪明地使用Perl的模式匹配功能。

关于Perl的模式匹配，要知道的最重要的一点是：什么时候不要使用模式匹配。

## 让Perl工作

有点急躁的人第一次学习正则表达式时，往往认为模式匹配中的所有一切都有问题。从大的方面来讲，这可能是对的，不过模式匹配不只是计算正则表达式。这有点像寻找你的车钥匙，要在丢钥匙的地方找，而不能为了看得更清楚就只在路灯下找。在实际生活中，我们都知道，在正确的位置找会比在错误的地方找高效得多。

类似的，要使用Perl的控制流来确定该执行哪些模式，而跳过哪些模式。正则表达式相当聪明，不过它只是像马一样聪明。如果它一下子看到太多的东西，就会不知所措。所以有时你必须为它戴上眼罩。例如，还记得前面的候选项例子吧：

```
/Gandalf|Saruman|Radagast/
```

这确实能像我们所说的那样做，不过没有想象中那么好，因为它会搜索字符串中的每一个位置查找每一个名字，然后才移到下一个位置。看过“指环王”的聪明读者可能会记得，在上面提到的这3个精灵中，Gandalf提到的最多，远远多于Saruman，而提到Saruman又远比Radagast频繁。所以通常更高效的做法是使用Perl的逻辑操作符来完成候选项处理：

```
/Gandalf/ || /Saruman/ || /Radagast/
```

这也是另外一种消除引擎“最左”策略的方法。只有当找不到Gandalf时才会搜索Saruman。另外只有当完全没有Saruman时才会搜索Radagast。

这不仅可以改变搜索的顺序，有时还能让正则表达式优化器更好地工作。通常，搜索单个串比同时搜索多个串要更容易优化。类似地，锚定的搜索如果不是太复杂，通常都可以优化。

控制流的控制并不仅限于||操作符。通常可以在语句级完成控制。一定要考虑首先筛选出最常见的情况。假设你在写一个循环来处理一个配置文件。很多配置文件中大部分都是注释。通常最好的办法是先删除这些注释和空行，然后再完成具体的处理，尽管这些具体处理也会在处理过程中删除注释和空行，但先删除再处理的做法会更高效：

```
while (<CONF>) {  
    next if /^#/;  
    next if /^\\s*(#|$)/;  
    chomp;  
    munchabunch($_);  
}
```



即使不是为了更高效，通常也需要把普通的Perl表达式换成正则表达式，因为你可能想做一些处理，而这些处理在正则表达式中是不可能的（或者很难做到），如打印。下面是一个有用的数字分类程序：

```
warn "has nondigits"      if /\D/;
warn "not a natural number" unless /^~\d+$/;          # 拒绝 -3
warn "not an integer"     unless /^~?\d+$/;           # 拒绝 +3
warn "not an integer"     unless /^[+-]?\d+$/;
warn "not a decimal number" unless /^~?\d+\.\d*$/;    # 拒绝 .2
warn "not a decimal number" unless /^~?(?:\d+(?:\.\d*)?|\.\d+)$/;
warn "not a C float"
    unless /^(([+-]?)(?=\d|\.\d)\d*(\.\d*)?([Ee]([+-]?\d+)))?$/;
```

我们可以把这一节延伸得更长，不过，实际上以上内容正是这整本书要介绍的。随着我们进一步深入，你会看到更多有关Perl代码与模式匹配相互作用的例子。尤其是后面的“编程化模式”一节（当然，先读一读相关资料也很好）。

## 变量内插

使用Perl的控制流机制来控制模式匹配也存在局限性。主要困难在于这是一种“全有或全无”的方法。要么运行模式，要么不运行。有时你知道你想要的模式的大致轮廓，不过还希望能够参数化。变量内插提供了这种能力，就像参数化子例程一样，可以对子例程的行为有更多影响，而不只是确定是否调用它（关于子例程的更多内容见下一章的介绍）。

内插的一个好处是能够提供一点抽象，还能提高一点可读性。利用正则表达式，你可以写出更为简洁的代码：

```
if ($num =~ /^[-+]?\d+\.\d*$/) { ... }
```

不过，如果写为以下代码，能更清楚地表达你的意思：

```
$sign = '[-+]?';
$digits = '\d+';
$decimal = '\.?';
$more_digits = '\d*';
$number = "$sign$digits$decimal$more_digits";
...
if ($num =~ /^$number$/o) { ... }
```

我们将在这一章后面“生成模式”中更详细地讨论内插的这种使用。这里只是要指出可以使用/o修饰符来抑制重新编译，因为我们不希望在程序执行过程中改变\$number的值。现在这已经没有必要了，因为Perl能更聪明地处理这种问题，不过在比较早的代码中还可能看到这种用法。

另一个漂亮的小技巧是，可以把内部测试放在外面，使用字符串变量与一组已知字符串完成模式匹配：

```

chomp($answer = <STDIN>);
if ("SEND" =~ /\Q$answer/i) { say "Action is send" }
elsif ("STOP" =~ /\Q$answer/i) { say "Action is stop" }
elsif ("ABORT" =~ /\Q$answer/i) { say "Action is abort" }
elsif ("LIST" =~ /\Q$answer/i) { say "Action is list" }
elsif ("EDIT" =~ /\Q$answer/i) { say "Action is edit" }

```

这样一来，你的用户只要输入S、SE、SEN或SEND（任意大小写组合），都会完成“send”（发送）动作。要完成“stop”（停止），他至少要输入ST（或St、sT或者st）。

## 如果有反斜线

考虑双引号内插时，通常会同时想到变量内插和反斜线内插。不过我们前面提到，对于正则表达式，有两轮处理，前一轮内插处理会把大多数反斜线内插延迟到由正则表达式解析器来处理（稍后我们会介绍）。正常情况下，你不会注意到差别，因为Perl会努力隐藏这种差别。

实际上，由正则表达式解析器处理反斜线非常重要，因为只有正则表达式解析器知道哪个**\b**表示单词边界，而哪一个**\b**表示反斜线。假设你用/x修饰符在一个模式中搜索制表符：

```
($col1, $col2) = /(.*?) \t+ (.*?) /x;
```

如果Perl没有将**\t**的解释推迟到正则表达式解析器，**\t**就会变成空白符，正则表达式解析器会由于/x的存在而将它忽略，而且毫无知觉。不过Perl没有这么差劲，或者不会这样骗人。

不过，你可以自己骗自己。假设你抽出列分隔符，如下：

```

$colsep = "\t+"; # (双引号)
($col1, $col2) = /(.*?) $colsep (.*?) /x;

```

现在可就麻烦了，因为**\t**在到达正则表达式解析器之前会变成一个真正的制表符，而正则表达式解析器把空白符丢掉之后以为你在说**/(.\*?) + (.\*?) /**。哎呀，真糟糕！要修正这个问题，要避免使用/x或使用单引号。或者更好的办法是使用qr//（参见下一节）。

像这样处理的双引号转义只有命名字符和6个转换转义：**\N{CHARNAME}**、**\U**、**\u**、**\L**、**\l**、**\F**、**\Q**和**\E**。如果查看Perl正则表达式编译器的内部工作，你会发现有一些处理转义的代码，如**\t**处理为制表符，**\n**表示换行符等。不过，你找不到处理这6个转换转义的代码（我们在表5-9中列出了这些转义字符，因为人们希望能看到它们）。如果你想以某种方式将这些字符偷偷放在模式中而不完成双引号计算，这些字符将无法识别。如果藏入一个命名字符，则它会生成一个错误，因为创建字符串时的活动字符映射表需要解析一个名字对应哪个码点（这是因为Perl允许你创建定制的字符名别名，所以它不一定是标准集合。参见第29章的“字符名”一节）。

这些字符如何处理呢？嗯，可以使用单引号作为模式定界符，这样就能取消内插。在



`m'...`，`qr'...`和`s'...'...`中，单引号会抑制变量内插和转换转义的处理，就像单引号字符串中一样。例如，`m'\ufrodo'`不会找到frodo的首字母大写的版本。不过，由于在这个层次上不会处理“正常的”反斜线字符，所以`m'\t\d'`仍能匹配一个真正的制表符后面跟着任意的数字。

还有一种消除内插的办法，可以通过内插本身来抑制。如果有：

```
$var = '\U';  
/${var}frodo/;
```

可怜的frodo仍然不是首字母大写。即使你内插的模式看起来可能需要重新内插，但Perl不会因此为你重新完成这一轮内插。不能指望它正常工作，同样的，也不能指望下面的这个双引号内插正常工作：

```
$hobbit = "Frodo";  
$var = '$hobbit';    # (单引号)  
/${var}/;            #表示m'$hobbit'，而不是m'Frodo'.
```

下面给出另一个例子，从这里可以看到，大多数反斜线会由正则表达式解析器解释，而不是由变量内插处理。假设你有一个用Perl编写的简单的类*grep*小程序<sup>注14</sup>：

```
#!/usr/bin/perl  
$pattern = shift;  
while (<>) {  
    print if /$pattern/;  
}
```

如果把这个程序命名为*pgrep*，并用以下方式调用：

```
% pgrep '\t\d' *.c
```

你会发现，它会打印出所有包含一个制表符后面跟一个数字的C源文件中的所有行。你不用做任何特殊的处理，Perl就能意识到`\t`是一个制表符。如果Perl的模式只是双引号内插，你就必须做些特殊的处理。幸运的是，Perl模式并不只是双引号内插。这些模式会由正则表达式解析器直接识别。

真正的*grep*程序有一个`-i`开关，可以关闭区分大小写的匹配。不用为你的*pgrep*程序增加这样一个开关；它已经能够处理这一点，而无需任何修改。只需要为它传入一个更有趣的模式，其中包含一个内嵌的*/i*修饰符：

```
% pgrep '(?i)ring' LotR*.pod
```

现在它会搜索“Ring”、“ring”、“RING”等。在直接量模式时不常看到这个特性，因

---

注14：如果你以前不知道*grep*程序是什么，现在应该知道。几乎没有不支持*grep*的系统，我们相信*grep*是有史以来最有用的小程序（因为逻辑上我们并不认为Perl是一个小程序）。



为你完全可以直接写为/ring/i。不过，对于在命令行上传入的模式、在web搜索表单中输入的模式，或者内嵌在配置文件中的模式，这会是一个“救生圈”（既然我们说到了游泳圈（ring））。

## qr/PATTERN/修饰符引用正则表达式操作符

内插到模式中的变量必须在运行时完成变量内插，而不是编译时。原先这会显著降低执行速度，因为Perl必须检查你是否改变了变量的内容；如果是这样，它就必须重新编译正则表达式。如今，Perl要聪明得多，只有在内插长度超过10kB的模式时才会注意到/o选项的好处（这个选项如今几乎已经绝迹），它会告诉Perl只内插和编译一次：

```
print if /$pattern/o;
```

尽管这在我们的`pgrep`程序中运行得很好，但一般情况下并不能做到这一点。假设你有一组模式，希望在一个循环中分别匹配这些模式，可能如下所示：

```
for my $item (@data) {
    for my $pat (@patterns) {
        if ($item =~ /$pat/) { ... }
    }
}
```

你不能写为/`$pat`/o，因为在内循环中每次循环时`$pat`的含义都会改变。

对此的解决方法是使用`qr/PATTERN/msixpodual`操作符，通常这会简单地拼作`qr//`，原因很明显。这个操作符引用其`PATTERN`并作为一个正则表达式进行编译。`PATTERN`会以`m/PATTERN/`中同样的方式完成内插。如果用作定界符，就不会完成任何变量内插（或7个转换转义）。这个操作符返回一个特殊值，可以使用这个值取代相应模式匹配或替换中的等价直接量。例如：

```
$regex = qr/my.STRING/is;
s/$regex/something else/;
```

等价于：

```
s/my.STRING/something else/is;
```

所以对于上面的嵌套循环问题，首先使用一个单独的循环对模式进行预处理：

```
@regexes = ();
for my $pat (@patterns) {
    push @regexes, qr/$pat/;
}
```

或者同时使用Perl的`map`操作符：

```
@regexes = map { qr/$_/ } @patterns;
```

然后修改循环来使用这些预编译的正则表达式：

```
for my $item (@data) {  
    foreach $re (@regexes) {  
        if ($item =~ /$re/) { ... }  
    }  
}
```

现在当你运行匹配时，Perl没有必要为每一个if测试都创建一个编译的正则表达式，因为它会看到已经有一个编译的正则表达式。

qr//的结果甚至会内插到一个更大的匹配中，就好像它是一个简单的字符串一样：

```
$regex = qr/$pattern/;  
$string =~ /foo${regex}bar/; # 内插到更大的模式中
```

这一次Perl确实会重新编译模式，不过可以把多个qr//操作符链在一起变成一个操作。

之所以可以这样做，原因在于qr//操作符返回的是一种特殊类型的对象，这个对象重载了字符串串接方法，见第13章的介绍。打印返回值时会看到等价的字符串：

```
use v5.14;  
$re = qr/my.STRING/is;  
say $re; # v5.14中打印(?^usi:my.STRING)
```

^指出从默认的选项设置开始。这里有/u是因为可以使用Perl提供的“unicode\_strings”特性，因为你已经声明use v5.14。模式中启用了/s和/i修饰符，因为它们要提供给qr//。这里没有提到/x和/m，因为在开头脱字符指定的默认环境中禁用了这两个修饰符，脱字符要求从原先的修饰符开始，而不是当前修饰符。

如果将未知来源的字符串内插到一个模式中，你就要做好准备去处理正则表达式可能抛出的任何异常，因为有人可能会输入一个包含可怕“野兽”的字符串：

```
$re = qr/$pat/is; # 可能转义并“吃掉”你  
$re = eval { qr/$pat/is } || warn... # 把它抓到外面的笼子里
```

关于eval操作符的更多内容，参见第27章。

## 正则表达式编译器

在变量内插完成对字符串的处理后，终于轮到正则表达式解析器来理解你的正则表达式了。这里实际上不会有太大问题，除非小括号混乱，或者使用了没有任何意义的元字符序列。解析器会对你的正则表达式完成一个递归下降分析，如果解析，会把它转换为一种适合引擎解释的形式（参见下一节）。解析器中大多数有趣的处理都有关于优化正则表达式，让它运行得尽可能更快。我们并不打算解释这一部分。这是一个商业机密（有谣言说查看正则表达式代码会让你疯狂。希望这只是谣言）。

不过你可能想知道解析器具体如何处理你的正则表达式，如果你礼貌地问它，它会告诉你它的答案。通过声明`use re "debug"`，就可以检查正则表达式解析器如何处理你的模式（使用命令行开关`-Dr`也可以看到同样的信息，如果你安装Perl时使用了`-DDEBUGGING`标志完成编译，就可以使用这个命令行开关）。

```
#!/usr/bin/perl
use re "debug";
"Smeagol" =~ /^Sm(.*)[aeiou]l$/;
```

输出如下。可以看到，在执行之前，Perl会编译正则表达式，并为模式的各个部分指定含义：BOL表示行开头（`^`），REG\_ANY表示点号，依此类推：

```
Compiling REx "^Sm(.*)[aeiou]l$"
Final program:
 1: BOL (2)
 2: EXACT <Sm> (4)
 4: OPEN1 (6)
 6: STAR (8)
 7: REG_ANY (0)
 8: CLOSE1 (10)
10: ANYOF[aeiou][] (21)
21: EXACT <l> (23)
23: EOL (24)
24: END (0)
anchored "Sm" at 0 floating "l"$ at 3..2147483647 (checking anchored)
anchored(BOL) minlen 4
```

有些行对正则表达式优化器的结论做了小结。它知道字符串必须以“Sm”开头，所以没有必要做普通的从左到右扫描。它还知道字符串必须以一个“l”结尾，所以可以拒绝不是以一个“l”结尾的字符串。另外，它知道字符串必须至少4个字符长，所以可以忽略所有长度没有达到这个标准的字符串。它还知道各个常量字符串中出现次数最少的字符，这有助于在用study“研究过”的字符串中进行搜索（参见第27章中对study的介绍）。

然后它会继续跟踪如何执行这个模式：

```
Guessing start of match in sv for REx "^Sm(.*)[aeiou]l$" against "Smeagol"
Guessed: match at offset 0
Matching REx "^Sm(.*)[aeiou]l$" against "Smeagol"
 0 <> <Smeagol> | 1:BOL(2)
 0 <> <Smeagol> | 2:EXACT <Sm>(4)
 2 <Sm> <eagol> | 4:OPEN1(6)
 2 <Sm> <eagol> | 6:STAR(8)
                  REG_ANY can match 5 times
                  out of 2147483647...
 7 <Smeagol> <> | 8: CLOSE1(10)
 7 <Smeagol> <> |10: ANYOF[aeiou][](21)
                  failed...
 6 <Smeago> <l> | 8: CLOSE1(10)
 6 <Smeago> <l> |10: ANYOF[aeiou][](21)
                  failed...
```



```

5 <Smeag> <ol>      | 8: CLOSE1(10)
5 <Smeag> <ol>      | 10: ANYOF[aeiou][](21)
6 <Smeago> <l>       | 21: EXACT <l>(23)
7 <Smeagol> <>      | 23: EOL(24)
7 <Smeagol> <>      | 24: END(0)
Match successful!
Freeing REx: "^Sm(.*)[aeiou]l$"

```

如果沿着Smeagol中间的一串空白符一路向下看，可以看到引擎如何“飞越”这些字符，使.\*尽可能地贪婪，然后对其回溯，直到找到一种方式使模式的余下部分匹配。不过这是下一节要讨论的内容。

## 小引擎/能与不能?/

现在我们要告诉你Perl的小正则表达式引擎是怎么说的：“我认为我能，我认为我能，我认为我能。”

这一节中，我们会给出Perl的正则表达式引擎将模式与字符串匹配时所使用的规则。正则表达式引擎非常坚持，而且不辞劳苦。甚至在你认为它该停工的时候还在努力工作。除非确信再没有办法将模式与字符串匹配，否则引擎绝不会放弃努力。下面的规则解释了引擎会尽可能长地“认为它能”，直到它很清楚地知道它能或不能。对我们的引擎来说，问题在于它的任务并不只是简单地沿着铁轨把火车开上山。它必须搜索一个（有可能）非常复杂的可能性空间，记录它去过哪里，另外哪些地方还没有去过。

引擎使用了一个非确定性的有限状态自动机（Nondeterministic Finite-state Automaton，NFA）来查找匹配，不要把NFA与NFL混为一谈（NFL表示一个不确定的足球联盟）。这表示它会记录做过哪些尝试，而哪些还没有尝试过，如果一条路走不通，则它会后退，尝试其他的路。这称为回溯（backtracking）（嗯，抱歉，这个词不是我们发明的，真的）。引擎能够在一点上尝试上百万个子模式，如果这些都失败，则放弃，向开始方向后退一步，然后在另一个点上再尝试上百万个子模式。引擎并不算非常聪颖，不过确实很执着，而且非常全面。如果你是个精明的人，则可以为引擎提供一个高效的模式，不要让它做太多傻乎乎的回溯。

如果有人提到这样的说法“正则表达式会选择最左最长的匹配”，这表示Perl通常更倾向于最左匹配而不是最长匹配。不过引擎并不认为它有任何“倾向性”，实际上它根本没有做更多考虑，只是在一个一个地尝试。总的倾向性是很多单个的没有关联的选择共同作用所得到的自然行为。这些选择包括：<sup>注15</sup>

### 规则1

引擎从字符串左边尽可能远的位置匹配，使得整个正则表达式基于规则2匹配。

注15：如果正则表达式优化器在起作用，其中一些选择可能会被忽略，这等价于小引擎通过特定的隧道穿过小山。不过，在这个讨论中，我们假设不存在优化器。

引擎从第一个字母前的位置开始，尝试从那里匹配整个模式。当且仅当引擎消费完整个字符串之前能够到达模式末尾，则认为整个模式匹配。如果模式确实匹配，就会立即退出。它不会继续寻找一个“更好”的匹配，尽管模式可能会以很多不同的方式匹配。

如果在字符串的第一个位置无法匹配模式，则它会承认暂时失败，并转向字符串中的下一个位置，即第一个和第二个字符之间的位置，然后再次尝试所有可能性。如果成功，则停止。如果失败，则它会继续在字符串中的后续位置查找匹配。除非已经尝试了所有可能性，即字符串中的每一个位置都无法匹配整个正则表达式（包括最后一个字符后面的位置），此时才认为整个模式匹配失败。

一个 $n$ 字符的字符串实际上提供了 $n + 1$ 个可以匹配的位置。这是因为，匹配的开始和结束位置都在字符串的字符之间（或两端）。这个规则有时会让人有些莫名其妙，他们可能会写类似`/x*/`的模式来匹配0个或多个“x”字符。如果对一个类似“fox”的字符串尝试匹配这个模式，它不会找到“x”。实际上，它会立即匹配“f”前的空串，然后不会再继续查找。如果你希望它匹配一个或多个x字符，需要使用`/x+/`。请参见规则5中关于量词的介绍。

作为这个规则的一个推论，任何能匹配空串的模式都肯定能够在字符串中的最左位置匹配（前提是没有0宽度断言禁止这一点）。

## 规则2

引擎遇到一组候选项时（由`|`符号分隔），不论在最高层次还是在当前分组层次，它会从左到右尝试这些候选项，得到第一个成功的匹配时停止，这也会完成整个模式的成功匹配。

如果任何一个候选项基于规则3匹配，则候选项集合与字符串匹配。如果所有候选项都不匹配，它会回溯到调用这个规则的上一级规则，通常是规则1，不过也可能是规则4或规则6（如果在一个分组中）。然后那个规则会查找一个新位置，并在这个新位置再次应用规则2。

如果只有一个候选项，它要么匹配，要么不匹配，规则2仍适用（没有0个候选项的说法，因为空串总会匹配）。

## 规则3

如果候选项中所列的每一项都能根据规则4和规则5按顺序匹配，那么这个特定的候选项匹配（相应的，可以满足整个正则表达式）。

项可能包括一个断言（见规则4），或者包括一个量化的原子（见规则5）。能够选择匹配方式的项会有一种从左到右的“长幼顺序”。如果这些项不能按顺序匹配，引擎会根据规则2回溯到下一个候选项。

正则表达式中必须顺序匹配的项并没有由某个语法项分隔，它们只是按匹配所应遵



循的顺序并列放置。如果要匹配`/^foo/`，实际上要求这4项一个接一个地匹配。第一个是一个0宽度断言，要根据规则4匹配，另外3个是普通的字符，必须根据规则5依次与自身匹配。

由于有一个从左到右的“长幼顺序”，这说明在类似下面的一个模式中：

`/x*y*/`

`x*`会选择一条路匹配，然后`y*`尝试它的所有可能性。如果失败，`x*`再走向它的下一个选择，让`y*`再尝试所有可能的匹配。依此类推。右边的项会“更快地更替”，这个说法是从多维数组借用的。

#### 规则4

如果一个断言在当前位置不匹配，引擎会回溯到规则3，采用不同的选择再次尝试长幼顺序中更靠前的项。

有些断言会更有趣一些。Perl支持很多正则表达式扩展，其中一些是0宽度断言。例如，正前瞻`(?=...)`和负前瞻`(?!...)`并不具体匹配任何字符，而只是假想地断言：假设我们做出尝试，...表示的正则表达式将会（或不会）在这一点匹配<sup>注16</sup>。

#### 规则5

只有当原子本身能匹配量词所允许的次数时，量化的原子才会匹配（原子本身按照规则6匹配）。

不同的量词要求不同的匹配次数，其中大多数量词都会限定一个匹配次数的范围。多次匹配必须连续匹配。也就是说，它们在字符串中必须是相邻的。一般认为一个非量化的原子也有一个要求只匹配一次的量词（也就是说，`/x/`等同于`/x{1}/`）。对于一个量化原子，如果在当前位置该原子的数量不在允许的范围内，引擎会回溯到规则3，采用不同的选择再次尝试长幼顺序中更靠前的项。

量词包括`*`，`+`，`?`，`*?`，`++`，`??`，`*+`，`++`，`?+`和各种大括号形式。如果使用`{COUNT}`形式，那么别无选择，原子必须匹配指定的次数，否则无法匹配。不过，原子可以匹配多个量词，引擎会记录所有这些选择，从而能够在必要的情况下回溯。不过，这样一来会有一个问题：首先应当尝试其中的哪一个选择呢？有人可能从最大的匹配次数开始，逐步减少（倒着数），或者先选择最小的匹配次数，依次增加（正着数）。

传统的量词（没有末尾的问号）会指定贪婪匹配。也就是说，它们力图匹配尽可能多的字符。要找到最贪婪的匹配，引擎必须非常谨慎。不好的猜测很可能代价很昂贵，所以引擎不会从最大次数向下倒数，毕竟这个最大次数可能非常大，可能导致

---

注16：事实上，引擎确实会尝试。它会回退到规则2，测试子模式，然后清除已经“吞掉”字符串中多少字符的记录，只返回子模式的成功或失败，作为断言的结果值（不过，它能记住捕获的子串）。



上百万不好的猜测。引擎的做法要稍稍聪明一些：它首先正着数，查找字符串中实际上有多少匹配的（连续）原子，然后使用这个实际的最大数作为第一个选择（它还会记住所有较短的选择，以防万一这个最长的选择无法匹配）。然后假设最长的选择是最佳的，（在最后）尝试匹配模式的其余部分。如果最长的选择未能保证模式的其余部分匹配，它会回溯，并尝试下一个最长的选择。

例如，如果写为`/. *foo/`，在查找“foo”之前，会尝试匹配最大数目的“任意”字符（由点号表示），直到到达行尾；在这里“foo”不会匹配（它也无法匹配，因为字符串末尾没有为它留出足够的空间），引擎会一次回退一个字符，直到找到一个“foo”。如果一行中有多个“foo”，它会在最后一个“foo”停下来，因为这会是它回溯时遇到的第一个“foo”。如果使用某个特定长度的`. *`能保证整个模式成功，此时引擎知道它可以扔掉所有其他较短的`. *`（如果当前“foo”没有成功则有可能使用这些较短的选择）。

通过在贪婪量词后面加一个问号，可以把它变成一个“节俭”量词，第一次尝试时会选择最小的数目。所以如果写为`/. *?foo/`，`. *?`首先会尝试匹配0个字符，然后是1个字符，然后两个，依此类推，直到可以匹配“foo”。并不是向后回溯，它会向前回溯，所以最后会找到一行上的第一个“foo”而不是最后一个。

## 规则6

每个原子都会根据其类型的指定语义来匹配。如果原子不匹配（或者尽管本身能匹配，但不能保证模式的其余部分匹配），引擎会回溯到规则5，并尝试原子数量的下一个选择。

原子根据以下类型匹配：

- 小括号里的正则表达式`(...)`，匹配正则表达式（由`...`表示）根据规则2所匹配的字符串。因此，小括号相当于一个用于量化的分组操作符。裸小括号还有一个副作用，可以在一个组引用中捕获匹配的子串，以备以后使用，这个组引用也称为反向引用（backreference）。可以使用`(?:...)`来禁止这个副作用，这样一来，它只有分组语义，而不会把任何结果存储在`$1`，`$2`等变量中。加小括号的原子（和断言）还可以有其他形式，参见这一章后面的内容。
- 点号匹配任意字符，可能除换行符以外。
- 中括号中的字符列表（中括号字符类）匹配这个列表指定的任意一个字符。
- 加反斜线的字母匹配一个抽象序列，通常是一个特定的字符，或者是一组字符中的一个，如表5-3所列。
- 任何其他加反斜线的字符匹配该字符。
- 以上没有提到的字符均匹配自身。

看上去相当复杂，不过基于这些规则有一个结果：对于一个量词或候选项提供的每一组选择，引擎就像分别有一个可以转动的旋钮。它会不断转动这些旋钮，直到整个模式匹配。前面的规则只是说明了引擎会以怎样的顺序转动这些旋钮。假设引擎更倾向于最左匹配，这只是表示它会用最慢的速度转动起始位置旋钮（对应最左位置）。而回溯只是撤销刚才的转动，去转动长幼顺序中更靠前的旋钮（也就是说变化更慢的旋钮）。

下面是一个更具体的例子。这个程序要检测两个连续的单词是否有相同的结尾和开头：

```
$a = "nobody";  
$b = "bodysnatcher";  
if (" $a $b" =~ /^(\w+)(\w+) \2(\w+)$/ ) {  
    say "$2 overlaps in $1-$2-$3";  
}
```

这会打印出：

```
body overlaps in no-body-snatcher
```

你可能以为\$1由于贪婪性首先会捕获整个“nobody”。实际上也确实是这样，不过只是一开始会这样。一旦这样做，就没有更多字符可以放在\$2中，而由于+量词的存在，所以\$2中需要有字符。因此，引擎会回退一步，\$1只好不情愿地放弃一个字符给\$2。这一次，空格字符可以成功匹配，不过接下来它看到\2，这表示一个小小的“y”。而字符串中的下一个字符不是“y”，而是一个“b”。这会让引擎就这样一个字符一个字符地后退，最终要求\$1把整个body都让给\$2。就好像有人身保护令一样，把body从非法的拘禁中豁免出来。

实际上，如果重叠本身是双写的结果，就不会这样做，如单词“rococo”和“cocoon”中存在着双写。以上算法会确定重叠的字符串\$2必然是“co”而不是“coco”。不过，我们可不想要一个“rocococoon”，而只想要一个“rococoon”。在这里你可以比引擎更聪明（这种机会不多）。为\$1部分增加一个最小匹配量词，可以得到一个更好的模式`/^(\w+?) (\w+) \2(\w+)$/`，它将得到我们想要的结果。

关于各种不同正则表达式引擎的优缺点，更详细的讨论请参见Jeffrey Friedl编写的《Mastering Regular Expressions》。对于你想用perl处理的很多日常问题，Perl的正则表达式引擎都很适用，如果你给它多一点尊重和理解，甚至还能处理那些不太常见的问题。

## 非传统模式

### 环视断言

有时你需要稍稍偷看一眼。有4个正则表达式扩展可以帮你做到这一点，我们把它们称为环视断言（lookaround assertions），因为它们允许你以某种方式环视，这只是假设性的，并不真正匹配任何字符。这些断言只是确认如果我们尝试某个模式，它会（或不会）匹



配。引擎会为我们得出这个结论，它会具体尝试匹配这个假想的模式，然后假装没有匹配（如果能匹配的话）。

如果引擎从字符串的当前位置向前看，我们称之为一个前瞻断言（lookahead assertion）。如果向后看，则称之为后顾断言（lookbehind assertion）。前瞻模式可以是任何正则表达式，不过后顾模式只能是固定宽度，因为它们必须知道从哪里开始假想的匹配。

尽管这4个扩展都是0宽度断言，相应地不会消费任何字符（至少，不会正式消费字符），不过如果提供额外的一层捕获小括号，实际上它确实可以捕获子串。

### `(?=PATTERN)` (正前瞻)

引擎遇到`(?=PATTERN)`时，它会在字符串中向前看（前瞻），确保这个`PATTERN`出现。如果你还记得，在前面删除重复单词的代码中，我们必须写一个循环，因为模式每次“吞掉”的东西太多：

```
$_ = "Paris in THE THE THE THE spring.";
# 删除重复出现两次的单词(和三次(以及四次...))
1 while s/\b(\w+) \1\b/$1/gi;
```

只要听到“吞掉太多”这种说法，就该想到“前瞻断言”（嗯，几乎总会想到）。通过向前偷偷看一眼，而不是直接掠取第二个单词，可以写出只需一趟处理的重复单词删除代码（而不需要循环），如下：

```
s/ \b(\w+) \s (?= \1\b ) //gxi;
```

当然，这还不太对，因为它会破坏类似“The clothes you DON DON't fit.”这种合法的句子。

前瞻断言可以用来实现重叠匹配。例如，

```
"0123456789" =~ /(\d{3})/g
```

只返回3个字符串：012，345和678。通过用一个前瞻断言来包装捕获组：

```
"0123456789" =~ /(?:\d{3})/g
```

现在能得到所有可能的字符串，包括012，123，234，345，456，567，678和789。这是可行的，因为这个聪明的断言偷偷地向前获取了前面的内容，并填入它的捕获组，不过作为一个前瞻，它会否认自己所做的，而不真正消费任何字符。引擎看到由于有`/g`而需要再次尝试时，它会从上一次尝试的位置后退一个字符。

### `(?!PATTERN)` (负前瞻)

引擎遇到`(?!PATTERN)`时，它会在字符串中向前看（前瞻），确保这个`PATTERN`不出现。要修正前一个例子，可以在正断言后面加一个负前瞻断言，消除冲突的情况：

```
s/ \b(\w+) \s (?= \1\b (?! '\w'))//xgi;
```



最后这个\w是必要的，可以用来避免与双引号字符串末尾的单词发生冲突。这一步还可以走得更远，因为本章前面我们有意地使用了一个例子“that that particular”，希望程序不要盲目地帮我们“修正”这个原本正确的短语。所以可以增加一个负前瞻候选项，提前禁止修正这个“that”（相应地还展示了可以使用任意对小括号对候选项分组）：

```
s/ \b(\w+) \s (?= \1\b (?! '\w | \s particular))//gix;
```

现在我们知道这个“that that particular”短语是安全的。遗憾的是，葛底斯堡演说（1863年美国总统林肯所作的关于“民治，民有，民享”的演说）还是会被错误地“修正”。所以我们再增加一个例外情况：

```
s/ \b(\w+) \s (?= \1\b (?! '\w | \s particular | \s nation))//igx;
```

这样一来，情况开始有点失控了。所以下面我们建立一个正式的例外情况列表，将利用一个巧妙的内插技巧：使用\$"变量从而用|字符分隔候选项：

```
@thatthat = qw(particular nation);
local $" = '|';
s/ \b(\w+) \s (?= \1\b (?! '\w | \s (?: @thatthat )))//xig;
```

### (?<=PATTERN) (正后顾)

引擎遇到 (?<= PATTERN)时，它会在字符串中向后看（后顾），确保PATTERN已经出现。我们的例子还存在一个问题。尽管现在它允许正直的亚伯（林肯的昵称）讲类似“that that nation”之类的话，不过这样一来，“Paris, in the the nation of France”之类的句子也会被接纳。可以在我们的例外情况列表前面增加一个正后顾断言，确保只对真正的“that that”应用@thatthat例外。

```
s/ \b(\w+) \s (?= \1\b (?! '\w | (?<= that) \s (?: @thatthat )))//ixg;
```

没错，现在变得相当复杂了，不过正是因为这一点，这一节才取名叫“非传统模式”。如果你认为我们做得还不够，还需要把模式变得更复杂，则可以明智地使用注释和qr//，这会帮助你保持清醒，或者至少能更清醒一些。

或者可以考虑使用\K，哄骗引擎正式匹配从哪里开始。前面的模式就相当于对模式正式部分的一种后顾，不过会对它从左向右扫描。如果需要一个可变宽度的后顾（这是引擎做不到的，或者至少不打算这么做），\K就尤其有用。

### (?<!PATTERN) (负后顾)

引擎遇到(?<! PATTERN)时，它会在字符串中向后看（后顾），确保PATTERN没有出现。这一次我们举一个相当简单的例子。来看一个古老的拼写规则“I一般要放在E前面，但在C后面时例外，这种情况下I要放在E后面”。在Perl中，这个规则要写为：

```
s/(?<!c)ei/ie/g
```

必须仔细考虑是否希望处理这些例外情况（例如，“weird”就拼为weird，把它拼作“wierd”时才怪异）。

## 占有组

在“小引擎（能与不能）”一节中已经介绍过，引擎在处理模式时通常会回溯。可以通过一组特定的选择来阻止引擎回溯，即创建一个非回溯子模式（nonbacktracking subpattern）。占有组形式为`(?>PATTERN)`，它的工作与简单的非捕获组`(?:PATTERN)`很类似，只不过一旦`PATTERN`找到一个匹配，它就会禁止对该子模式中的任何量词或候选项回溯（因此，在一个不包含量词或候选项的`PATTERN`中使用占有组是没有意义的）。要让它改变心意，唯一的办法就是回溯到这个子模式之前的某个位置，然后从左重新进入子模式。

这有点像买车。几轮讨价还价之后，你最后报价：“这是我给的最后价格，要么成，要么不成。”如果他们不接受，你也不会再回过去继续砍价。实际上，你会退出门外。可能会去寻找另一家汽车经销商，然后再开始砍价。你可以再谈价钱，不过这只是因为你在一个不同的上下文中再次进入了这个非回溯模式。

对于喜欢Prolog或SNOBOL的人来说，可以把它想成是一个有作用域的截断（cut）或防卫（fence）操作符。

考虑`"aaab" =~ /(?:a*)ab/`，`a*`首先匹配3个`a`，然后放弃其中一个，因为后面需要最后这个`a`。这就是子组为了保证整个匹配成功，牺牲了自己想要的一些利益（这有点像汽车销售人员让你再多给一些钱，因为你害怕不成交）。与此相反，`"aaab" =~ /(?:>a*)ab/`中的子模式不会放弃它捕获的内容，尽管这个行为会导致整个匹配失败（就像歌中所唱的，你要知道什么时候坚持，什么时候放弃，什么时候安静地走开）。

尽管`(?:> PATTERN)`会对于改变模式的行为很有用，不过它的主要作用是加快某些匹配的失败，因为你知道它们肯定会失败（除非能立即成功）。引擎可能花相当长的时间才能得出匹配失败，特别是对于嵌套的量词。下面的模式几乎立刻就会成功：

```
$_ = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab";  
/a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*[Bb]/;
```

不过，成功并不是问题。失败时才有问题。如果从字符串末尾删除最后一个“b”，这个模式可能要运行很多很多年才会最终失败。甚至很多很多个世纪。实际上，可能是数百亿年<sup>注17</sup>。通过观察这个模式，如果字符串末尾没有“b”，你就会立即知道模式不会成功，不过正则表达式优化器还不够聪明（起码写这本书时还没有这么聪明），无法看出无论如

---

注17：实际上还会更多，数量级会达到10的24次方。我们并不能确切地知道究竟需要多长时间。

我们不会一直等到它失败的那一时刻。毕竟，在宇宙消亡之前你的计算机可能早已经崩溃了，而这个正则表达式可能比宇宙消亡需要的时间还要长。



何/[Bb]/都不会匹配。不过，如果你给它一个提示，就能让它快速失败，另一方面，如果本来能够匹配成功，则这个提示不会影响它的成功：

```
/(>a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a*a)[Bb]/;
```

对于一个更实际的例子（希望如此），假设一个程序要一次读入一个段落，只显示延续的行，延续行用末尾的反斜线指定。下面的例子就采用了这种行延续约定，取自Perl的*Makefile*。

```
# perl可用之前,
# 利用变量替换建立的文件
sh = Makefile.SH cflags.SH config_h.SH makeaperl.SH makedepend.SH \
    mkdir.SH myconfig.SH writemain.SH
```

可以这样写你的小程序：

```
#!/usr/bin/perl -00p
while ( /( (.+) ( (?<=\\) \n .* )+ ) /gx) {
    say "GOT $.: $1\n" ;
}
```

这是可以的，不过这样相当慢。因为引擎会从行尾一次回溯一个字符，缩减\$1中的字符串。这样很没有意义。而且如果改写模式，去掉多余的捕获，这样也帮助不大。使用以下模式：

$$(.+(?:(?<=\\)\\n.*)+)$$

会快一点，不过提高不大。这正是非回溯子模式能够大展身手的时候。下面这个模式：

$$((?>.+)(?: (?<=\\)\\n.*)+)$$

可以完成同样的事情，不过会快一个数量级，因为它不会浪费时间来回溯搜索根本没有的东西。

如果用(?:...)或者甚至简单的(...)不能成功，用(?>...)也不会成功。不过，如果会失败，最好快速失败，然后重新开始你的人生。

顺便说一句，由于我们的例子只包含一个量词，`(?>.+)`可能比写为`.++`更简洁。

## 编程化模式

大多数Perl程序都会采用一种命令式（也称为过程式）编程方式，就像用一种容易遵循的顺序下达的一系列单个命令：“炉子预热，混合，上釉，加热，冷却，交给顾客”。有时还会更复杂，可能会加一些函数式编程（“比你预想的要多上一点釉，甚至还可以递归地处理”），或者再加一点面向对象技术（“不过请装鱼类对象”）。通常这些编程方式会组合使用。



不过正则表达式引擎采用了一种完全不同的方法来解决问题，这种方法更应算是一种声明式方法。你要用正则表达式的语言描述目标，引擎则实现所需的逻辑来达到你的目标。尽管逻辑编程语言（如Prolog）没有另外3种编程方式那么声名显赫，不过它比你想象的更常用。如果没有`make(1)`或`yacc(1)`，甚至无法构建Perl，这两种语言（如果不算是纯粹的声明式语言）至少可以认为是综合了命令式和逻辑编程的混合语言。

Perl中也可以这样做，可以采用一种比以往更容易的方式将目标声明和命令式代码混合在一起，取二者所长。你可以通过编程构建最终提交给正则表达式引擎的字符串，从某种意义上说，这就是创建一个程序，它将动态地写一个新程序。

通过`/e`修饰符还可以提供普通的Perl表达式作为`s///`的替换部分。这就允许你在每次模式匹配时通过执行一些代码来动态地生成替换字符串。

更巧妙的是，可以使用`(?{ CODE })`扩展将一些代码插入到模式中间你希望的位置，引擎在它复杂的回溯“舞蹈”中每次向前或向后遇到这个代码时都会执行。

最后，可以使用`s///ee`或`(??{ CODE })`增加另外一层间接性：执行这些代码段的结果本身将重新计算，以备将来使用，这会即时地动态创建一些程序和模式。

## 生成模式

据说<sup>注18</sup>，写程序的程序是世界上最幸福的程序。在Jeffrey Friedl的《Mastering Regular Expressions》一书中，最后的例子展示了如何编写程序来生成一个正则表达式，以确定一个字符串是否遵循RFC 822标准；也就是说，这个字符串中是否包含与标准兼容的合法的邮件首部。所生成的模式有几千个字符长，读起来可能像纯二进制的崩溃转储信息一样“容易”。不过Perl的模式匹配工具并不在意这一点；它会毫不费力地编译这个模式，甚至更有意思的是，它能非常快速地执行匹配，实际上，相对于很多有复杂回溯需求的短模式，这个长模式的执行反而要快得多。

这是一个非常复杂的例子。之前我们曾基于各个分量来构造一个`$number`模式，那是一个用相同技术实现的非常简单的例子（参见前面的“变量内插”一节）。不过，为了展示这种方法（即通过编程来生成模式）的强大能力，下面考虑一个复杂程度居中的问题。

假设你想抽取所有遵循某个元音（`vowel`）－辅音（`consonant`）序列的单词；例如，“`audio`”和“`eerie`”都遵循VVCVV（元音－元音－辅音－元音－元音）模式。尽管描述哪些是辅音或哪些是元音很容易，但是你肯定不想反复输入这些字母。即使是这样一个简单的VVCVV模式，就需要输入类似下面的模式：

---

注18：这句话来自著名的UNIX哲学家Andrew Hume。

```
^[aeiouy][aeiouy][cbdfghjklmnpqrstvwxyz][aeiouy][aeiouy]$
```

更通用的程序能接受类似“VVCVV”的字符串，并通过编程来生成以上模式。为了更为灵活，它还可以接受类似“audio”的单词作为输入，用这个单词当作模板来推导出“VVCVV”，再由此生成上面的那个长模式。听上去很复杂，不过实际上并不太复杂，因为我们会让程序来为你生成模式。下面是一个简单的*cvmap*程序，它能完成所有这些工作：

```
#!/usr/bin/perl
$vowels = "aeiouy";
$cons = "cbdfghjklmnpqrstvwxyz";
%map = (C => $cons, V => $vowels);    # C和V的初始映射

for $class ($vowels, $cons) {        # 现在对于每个类型
    for (split //, $class) {          # 得到该类型的各个字母
        $map{$_} .= $class;          # 将字母映射回类型
    }
}

for $char (split //, shift) {        # 对于模板单词中的各个字母
    $pat .= "[$map{$char}]";          # 增加适当的字符类
}

$re = qr/^{pat}$/i;                  # 编译模式
say "REGEX is $re\n";                 # 调试输出
@ARGV = ("/usr/share/dict/words")     # 选择一个默认字典
    if -t && !@ARGV;

while (<>) {                           # 现在努力处理输入
    print if /$re/;                   # 打印匹配的行
}
```

有趣的内容都包含在%map变量中。它的键是字母表的各个字母，相应的值是其类型的所有字母。我们还放入了C和V，所以指定“VVCVV”或“audio”都是可以的，都能匹配得出“eerie”。给程序提供的参数中的各个字符将用来抽取正确的字符类，并增加到模式中。一旦创建模式并用qr//编译，匹配就能非常迅速地运行（即使是一个非常长的模式）。

如果对“fortuitously”运行这个程序会得到以下结果：

```
% cvmap fortuitously /usr/dict/words
REGEX is (?i-xsm:^[cbdfghjklmnpqrstvwxyz][aeiouy][cbdfghjklmnpqrstvwxyz][cbdfghjklmnpqrstvwxyz][aeiouy][aeiouy][cbdfghjklmnpqrstvwxyz][aeiouy][aeiouy][cbdfghjklmnpqrstvwxyz][cbdfghjklmnpqrstvwxyz][aeiouy][cbdfghjklmnpqrstvwxyz])
carriageable
circuitously
fortuitously
languorously
marriageable
milquetoasts
sesquiquarta
```

```
sesquiquinta  
villainously
```

查看这个正则表达式，可以看到，尽管编程有点曲折有些枯燥，不过可以省去大量输入的烦恼。

## 替换计算

在一个 `s/PATTERN/CODE/e` 表达式中使用 `/e` 修饰符时（“e”表示表达式计算），替换部分会解释为一个 Perl 表达式，而不只是一个双引号字符串。这就像一个嵌套的 `do { CODE }`。尽管它看起来像是一个字符串，但实际上只是一个代码块，会与程序的其余部分在同一时间编译，这往往比替换的发生早得多。

可以使用 `/e` 修饰符构造替换字符串，而且构造的字符串可以有双引号内插所不允许的更奇特的逻辑。从下面的例子可以看出二者的不同：

```
s/(\d+)/$1 * 2/;      # 把"42"替换为"42 * 2"  
s/(\d+)/$1 * 2/e;     # 把"42"替换为"84"
```

下面会将摄氏度转换为华氏度：

```
$_ = "Preheat oven to 233C.\n";  
s/\b(\d+\.?\d*)C\b/int($1 * 1.8 + 32) . "F"/e; # 转换为451F
```

这种技术的应用是无止境的。下面是一个过滤器，它会原地修改其文件（就像一个编辑器），为行首的每一个数字增加100（而且行首的数字后面跟有一个冒号，我们会提前查看这个冒号，但不会真正匹配或替换这个冒号）：

```
% perl -pi -e 's/^(\\d+)(?:=:/100 + $1/e' filename
```

有时你可能希望做更多事情，而不只是在另一个计算中使用你匹配的字符串。有时你希望这个字符串本身就是一个计算，要用它自己的计算来得到替换值。在第一个 `/e` 之后每一个附加的 `/e` 都会要在要执行的代码外加一个 `eval`。下面的两行代码会做同样的事情，不过第一个更可读：

```
s/PATTERN/CODE/ee  
s/PATTERN/eval(CODE)/e
```

可以使用这个技术将简单标量变量的名字替换为它们的值：

```
s/(\$\\w+)/$1/eeg;    # 内插大多数标量值
```

由于这实际上是一个 `eval`，`/ee` 甚至可以找到词法作用域变量。再看一个更精巧的例子，为（非负）整数的简单算术表达式计算一个替换：

```
$_ = "I have 4 + 19 dollars and 8/2 cents.\n";  
s{ (  
    \d+ \s*      # 查找一个整数
```



```

        [+* / -]      # 和一个算术操作符
        \s* \d+       # 和另一个整数
    )
}{ $1 }eegx;         # 然后扩展$1, 运行这个代码
print;               # "I have 23 dollars and 4 cents."

```

与所有其他eval *STRING*类似, 编译时错误(如语法问题)和运行时异常(如除0)都将会捕获。如果有这些错误, \$@ (\$EVAL\_ERROR)变量会告诉你出错了。

## 匹配时代码计算

在大多数使用正则表达式的程序中, 外围程序的运行时控制结构会驱动逻辑执行流。你可能会写if语句或while循环, 或者建立函数或方法调用, 有时它们会调用一个模式匹配操作。即使使用s///e, 也是替换操作符在控制, 只在成功的匹配后才会执行替换代码。

利用代码子模式 (code subpatterns), 正则表达式和程序代码之间的正常关系会颠倒过来。匹配过程中, 引擎在对模式应用规则时, 可能会遇到一个形如(?{ CODE })的正则表达式扩展。如果触发, 这个子模式并不做任何匹配, 也不会环视。这是一个一定会“成功的”0宽度断言, 计算这个断言的目的只是为了得到它的副作用。引擎执行模式时如果需要经过这个代码子模式, 就会运行这个代码。

```
"glyph" =~ /.+ (?{ say "hi" }) ./x;    # 打印"hi"两次
```

引擎尝试按照这个模式来匹配glyph时, 它首先让.+吞掉所有这5个字母。然后打印“hi”。它发现最后一个点号时, 所有5个字母已经被吞掉, 所以它需要回溯到.+, 让它放弃其中一个字母。然后它再在模式中向前移, 在把h赋给最后一个点号并成功完成匹配之前, 它会停下来再打印一次“hi”。

CODE片段两边的大括号是为了提醒你这是一个Perl代码块, 它表现得就像一个词法作用域代码块。也就是说, 如果在其中使用my声明一个词法作用域变量, 它对这个代码块是私有的。不过, 如果使用local来局部化一个动态作用域变量, 可能与你期望的不一样。( ?{ CODE } )子模式会创建一个隐式的动态作用域, 这个动态作用域在其余的模式范围内都是合法的, 直到模式成功或者通过代码子模式回溯。可以这样来考虑, 到达这个块末尾时实际上并没有返回。相反, 它会向引擎做一个不可见的递归调用, 尝试匹配模式的其余部分。只有当这个递归调用完成, 才会从代码块返回, 解除局部变量的局部化<sup>注19</sup>。在下

---

注19: 熟悉递归下降解析器的人可能发现这种行为很让人困惑, 因为这些编译器只要找到任何结果就会从递归函数调用返回。正则表达式引擎不是这样做的, 它找出某个结果时, 会进入更深的递归层次(甚至退出一个小括号分组时也一样)。递归下降解析器最后成功时会处在最小的递归层次上(递归最浅), 而正则表达式引擎在模式最后成功时会处在最大的递归层次上(递归最深)。你可能会发现, 如果把模式想成是一个调用树的线图表示可能会有帮助。把这个图记在你的脑海里, 就能更好地理解本地变量的动态作用域行为(如果做不到, 也不会更糟糕)。

一个例子中，我们在模式开头包含了一个代码子模式，从而将*\$i*初始化为0。然后用.\*匹配任意数目的字符，不过我们在.和\*之间加入了另一个代码子模式，可以统计.匹配了多少次。

```
$_ = "lothlorien";
m/ (?{ $i = 0 })          # 设置 $i 为0
  (. (?{ $i++ }) )*       # 更新$i, 甚至在回溯之后
  lori                    # 强制回溯
/x;
```

正则表达式引擎会欣然处理，将*\$i*设置为0，然后让.\*得到字符串中的所有10个字符。遇到模式中的直接量lori时，引擎回溯，从.\*让出这4个字符。匹配之后，*\$i*仍是10。

如果希望*\$i*反映.\*实际上匹配了多少个字符，则可以在模式中使用动态作用域：

```
$_ = "lothlorien";
m/ (?{ $i = 0 })
  (. (?{ local $i = $i + 1; }) )* # 更新$i, 不管是否回溯
  lori
  (?{ $result = $i })           # 复制到非局部化的位置
/x;
```

这里我们使用local来确保*\$i*包含.\*匹配的字符数，而不考虑是否回溯。正则表达式结束之后*\$i*就会被忘掉，所以代码子模式(?{ \$result = \$i })会确保统计的次数将保存在\$result中。

作为成功匹配的一部分，特殊变量 $\$R$ （见第25章的介绍）包含执行的最后一个(?{ CODE })的结果。

可以使用(?{ CODE })扩展作为一个(? (COND) IFTRUE | IFFALSE)的COND。如果是这样，不会设置 $\$R$ ，而且可以省略条件两边的小括号：

```
"glyph" =~ /.+(? ( ?{ $foo{bar} gt "symbol" } ) .|signet)./;
```

这里我们测试 $\$foo\{bar\}$ 是否大于symbol。如果确实大于，则在模式中包含.；如果不是这样，则在模式中包含signet。稍稍展开一点，可以让它更可读：

```
"glyph" =~ m{
  .+                # 任意字符
  (?(?{
    $foo{bar} gt "symbol" # 如果
                           # 为true
  })
  .                # 匹配另一个“任意字符”
  |                # else
  signet           # 匹配signet
)
.                # 更多任意字符
}x;
```

如果`use re "eval"`有效，那么即使正则表达式会内插变量，也允许在其中包含`(?{ CODE })`子模式：

```
/(.*) (?{length($1) < 3 && warn}) $suffix/;    # 如果没有use re "eval"
                                                # 则错误
```

这通常是不允许的，因为它可能存在安全风险。尽管上面的模式可能无害，因为`$suffix`是无害的，不过正则表达式解析器无法区分字符串的哪些部分要内插，而哪些部分不内插，所以如果存在内插，它会干脆完全禁止代码子模式。

如果模式来自被污染（tainted）的数据，即使有`use re "eval"`也不允许模式继续匹配。

如果`use re "taint"`有效，而且正则表达式的目标是被污染的字符串，那么捕获的子模式（可能保存在编号变量中，或者在列表上下文由`m//`返回的值列表中）也将是污染的。如果针对污染数据的正则表达式操作并不是要抽取安全的子串，而只是为了完成其他转换，这个特性就会很有用。关于污染的更多内容请参见第20章。如果有这个`pragma`，则认为预编译的正则表达式（通常由`qr//`得到）没有任何内插：

```
/foo${pat}bar/
```

如果`$pat`是一个预编译的正则表达式，这是允许的，即使`$pat`包含`(?{ CODE })`子模式。

之前我们展示了`use re 'debug'`可以打印调试输出。还有一个更原始的调试方案，可以使用`(?{ CODE })`子模式打印匹配过程中到目前为止匹配的内容：

```
"abcdef" =~ /.+ (?{say "Matched so far: $&"}) bcdef $/x;
```

这会打印以下结果：

```
Matched so far: abcdef
Matched so far: abcde
Matched so far: abcd
Matched so far: abc
Matched so far: ab
Matched so far: a
```

可以看出，`.+`会捕获所有字母，然后在引擎回溯时逐个地放弃字母。

## 匹配时模式内插

可以在模式内部构造模式的各个部分。`(??{ CODE })`扩展允许你在模式中插入代码，这些代码将计算为一个合法的模式。这有点像`/$pattern/`，只不过现在可以在运行时生成`$pattern`。更确切地讲，可以在匹配时生成模式。例如：

```
/\w (??{ if ($threshold > 1) { "red" } else { "blue" } }) \d/x;
```

如果`$threshold`大于1，这等价于`/\wred\d/`，否则这就等价于`/\wblue\d/`。



可以在计算的代码中包含组引用，从刚匹配的子串得出模式（即使之后回溯时它们可能变得不再匹配）。例如，以下模式将匹配所有正读和反读都一样的字符串（也称为回文，这样的短语中间有一个“峰点”）：

```
/^ (.+) .? (??{quotemeta reverse $1}) $/xi;
```

可以如下平衡小括号：

```
$text =~ /( \(+ ) (.*?) (??{ "\\" x length $1 })/x;
```

这会匹配形如(shazam!)和(((shazam!)))的字符串，并把shazam!放在\$2中。遗憾的是，它不会注意到中间的小括号是否平衡。为此我们需要递归。

很幸运，你也可以建立递归模式。一种方法是利用一个已编译模式，其中使用(??{ CODE })指向模式自身。递归匹配很不规则，这与正则表达式不同。所有关于正则表达式的介绍都会指出：标准的正则表达式不能正确地匹配嵌套的小括号。这是没错的。另外还有一点也是正确的：Perl的正则表达式并不标准。下面的模式<sup>注20</sup>能匹配一组嵌套的小括号，不管它们嵌套多深：

```
$np = qr{
    \ (
    (?:
        (?> [^()]+ )      # 无小括号，不回溯
    |
        (??{ $np })      # 用匹配小括号分组
    )*
    \ )
}x;
```

可以这样使用来匹配一个函数调用：

```
$funpat = qr/\w+$np/;
"myfunfun(1,(2*(3+4)),5)" =~ /^$funpat$/; # 匹配!
```

## 条件内插

(?(COND)IFTRUE|IFFALSE)正则表达式扩展类似于Perl的?:操作符。如果COND为true，就会使用IFTRUE模式；否则，会使用IFFALSE模式。COND可以是一个组引用（表示为一个裸整数，没有\或\$）、环视断言或者代码子模式（参见本章前面的“环视断言”和“匹配时代码计算”）。

如果COND是一个整数，会把它当作一个组引用来处理。例如，考虑以下代码：

```
#!/usr/bin/perl
$x = "Perl is free.";
```

---

注20：需要说明，如果要在某个语句中使用某个变量，不能在同一个语句中声明这个变量。当然，完全可以提前声明。

```
$y = 'ManagerWare costs $99.95.';

foreach ($x, $y) {
    /^(\\w+) (?:is|(costs)) (? (2) (\\$\\d+) | \\w+) /; # (\\$\\d+) 或 \\w+
    if ($3) {
        say "$1 costs money."; # ManagerWare costs money.
    } else {
        say "$1 doesn't cost money."; # Perl doesn't cost money.
    }
}
```

在这里，*COND*为(2)，如果第二个组引用存在，这就是true。如果是这样，会在模式的这个位置上包含(\\\$\\d+)（并创建\$3捕获变量）；否则会使用\\w+。

如果*COND*是一个环视断言或代码子模式，将使用断言的真值来确定包含*IFTRUE*还是*IFFALSE*：

```
/[ATGC]+(? ( ?<=AA) G | C) $/;
```

这里使用一个后顾断言作为*COND*来匹配一个DNA序列，它以AAG结尾，或者以另外某个基组合和C结尾。

可以省略*IFFALSE*候选项。倘若省略了这个候选项，如果*COND*为true，就会正常地在模式中包含*IFTRUE*模式；不过如果这个条件不为true，引擎会转向模式的下一部分。

## 递归模式

在模式中引用一个捕获组时，这个组实际捕获的内容将用于反向引用。

```
/\\b (      \\p{alpha}+ ) \\s+   \\1    \\b /x # 编号反向引用
/\\b (      \\p{alpha}+ ) \\s+   \\g{1}  \\b /x # 候选项语法
/\\b (      \\p{alpha}+ ) \\s+   \\g{-1}  \\b /x # 相对反向引用
/\\b (?<word> \\p{alpha}+ ) \\s+   \\k<word> \\b /x # 命名反向引用
```

这4个例子使用反向引用来匹配同一个单词连续出现两次。不过有时你可能希望用相同的模式匹配两个不同的单词。为此要使用一种不同的语法：(?*NUMBER*)会调用一个编号组的模式，而(?&*NAME*)会调用一个命名组的模式（后者可以算是对&形式子例程调用的追忆）。

```
/\\b (      \\p{alpha}+ ) \\s+ (?-1)    \\b /x # "调用"编号组
/\\b (?<word> \\p{alpha}+ ) \\s+ (?&word) \\b /x # "调用"命名组
```

采用*NUMBER*形式，前面的负号会从当前位置由右向左统计组，所以-1表示调用前一个组；你不用知道它的绝对位置。另一方面，如果数字前面有一个加号，如(?+*NUMBER*)，就要从当前位置向前统计右边有多少个组。

甚至可以调用当前所在的那个组，这会导致匹配引擎递归处理。这很常见，我们经常希望这样做。下面是匹配平衡小括号的一种方法：

```
/ ( \ ( (? : [^() ]++ | (?1) ) *+ \ ) ) /x
```

由于整个模式用捕获小括号包围，所以可以将它们都省略，使用(?0)来调用“第0组”，这表示整个模式：

```
/ \ ( (? : [^() ]++ | (?0) ) *+ \ ) /x
```

在这里这是可以的，不过如果写一个qr//，而它要用在另外某个模式中，结果可能与你预想的不一樣。下面我们定义了一个正则表达式，它能匹配一个标识符以及后面平衡的小括号：

```
$funcall = qr/\w+ ( \ ( (? : [^() ]++ | (?-1) ) *+ \ ) ) /x
```

然后如下调用：

```
while (<>) {  
    say $1 if /^ \s* ( $funcall ) \s* ; \s* $/x;  
}
```

这样可以很方便地只把所需要的结果保留在\$1中。子规则是一种位置独立代码（Position Independent Code）：它不关心在整个模式中的绝对位置。

对于简单的模式来说，编号组没有问题，不过对于更复杂的模式，你会发现命名组更可读：

```
$funcall = qr/\w+ (?<paren> \ ( (? : [^() ]++ | (?&paren) ) *+ \ ) ) /x  
while (<>) {  
    say $+{func} if /^ \s* (?<func> $funcall ) \s* ; \s* $/x;  
}
```

实际上，随着解析问题的规模变得越来越大，使用命名组是使你保持清醒的唯一方法。

需要说明，子调用并不会把它的捕获结果返回给外层模式；它只是返回最终的匹配位置。通常这正是你想要的，因为它要控制副作用，不过有时你可能希望保留解析得到的结果。稍后将介绍更多有关内容。

谈到解析，你现在可能已经意识到，你已经拥有了实现真正的解析所需要的几乎所有工具。这里所说的“真正的解析”并不是指NFA和DFA适用的简单状态机，而是真正的递归下降解析。利用这一节和下一节介绍的特性，Perl模式完全等价于一个递归下降解析器。只需要再加一些特性，你就能很轻松地编写看上去非常类似于yacc文件的完整文法或巴科斯范式文法。

## 文法模式

文法通过递归地解析子模式来实现。由于捕获组可以作为子模式，所以它们可以用作文法中的“产品”。你已经知道如何调用，只是不知道如何单独定义全部产品。



很显然，如果只是将捕获组用作一个子模式，它是否用来完成某个匹配并不重要。实际上，在写一个文法时，你通常并不希望在第一次定义子模式时就调用；可能希望先定义所有子模式，以后再开始使用。你真正想要的是有一种方法能隔离模式的某一部分，可以单独地定义，而不执行。

(?(DEFINE)...)块就可以达到这个目的。从技术上讲，这实际上就是一个条件，类似于我们前面看到的(?(COND)...)条件。在这里，条件是DEFINE，结果通常是false<sup>注21</sup>。所以DEFINE块中放置的任何内容都肯定不会执行。

一些自作聪明的人可能会说，既然条件总是false，编译器会丢掉这些false条件中无用的代码。不过，老师会告诉你，你所说的这个策略只适用于可执行代码，而不适用于声明。DEFINE块中的组被认为是声明性的，所以它们不会被丢掉。DEFINE块中的组可以一直作为子模式，在同一个模式中的其他位置调用。

所以这就是我们放置文法的地方。可以把DEFINE块放在你喜欢的任何地方，不过通常人们会把它放在主模式前后。在这个块中，定义顺序并不重要。所以模式最后看起来应该是这样：

```
qr{
  (?&abstract_description_of_what_is_being_matched)

  (?(DEFINE)
    (?<abstract_description_of_what_is_being_matched>
      (?&component1)
      (?&component2)
      (?&component3)
      ...
    )
    (?<component1> ... )
    (?<component2> ... )
    (?<component3> ... )
    ...
  )
}x;
```

这个模式中可以执行的部分都在DEFINE块以外，它会调用顶层规则，这个顶层规则再调用所有其他规则。

看起来这不只是一个传统的文法，还是一个传统的程序。与正常模式从左到右的处理不同，这个模式现在具有自顶向下编程的一般形式，其中包含多个子例程，它们相互之间迭代和递归地调用。这个开发模型的重要性再强调也不为过，因为为抽象指定好名字是保证模式匹配易于开发、调试和维护的最重要的事情。这与普通编程并无不同。

很容易看出哪里还需要抽象。如果某个地方有重复，可能需要抽取出重复的功能，并指定

---

注21：没错，这是个技巧，而且是一个漂亮的技巧。否则，我们就得使用一个后缀{0}量词（像Ruby一样），这会影响可读性。

名字。如果指定一个便于记忆的名字，指出它用来做什么，这将有助于你组织和维护代码。如果以后你想修改这个代码，只需要在一处修改；子例程调用仍然采用原来的代码重用范式。子模式就是伪装的子例程调用。

这种模式匹配实在让人着迷，一旦掌握，非平凡模式看上去就不再像经典的正则表达式，而更像它们功能强大的兄弟：文法。你不用再写：

```
/\b(?:=[\p{Alphabetic}\p{Digit}])\X(?:=[\p{Alphabetic}\p{Digit}])\X
|['\x{2019}](?:[^\x{2014}]\p{dash})+(?!=[\p{Alphabetic}\p{Digit}])\X)/
```

相反，你可能更愿意像下面这样，作为一个文法模式来抽取单词：

```
$word_rx = qr{ (?&one_word)
  (? (DEFINE)
    (?<a_letter>          (?= [\p{Alphabetic}\p{Digit}] ) \X          )
    (?<some_letters>      (?: (?&a_letter) | (?&tick) | (?&dash) ) + )
    (?<tick>              ['\N{RIGHT SINGLE QUOTATION MARK}]          )
    (?<dash>              (?= [^\N{EM DASH}] ) \p{dash}                )
    (?<one_word>
      \b
      (?= (?&a_letter) )
      (?&some_letters)
      (?! (?&a_letter)
        | (?&dash)
      )
    )
  ) # 结束define块
}x;
```

顶层模式只是调用one\_word正则表达式子例程，它会完成所有工作。你可以使用这个模式来打印一个文件中的所有单词，每行一个单词，如下所示：

```
while (/( $word_rx)/) {
  say $1;
}
```

如这些例子所示，有时也可以把文法模式写为老式的非文法模式（不包含正则表达式子例程）。不过你肯定不想那么做，因为如此一来会过于困难，以至于无法读、写、调试和维护类似这样的复杂模式。不过除此之外，文法还能赋予你更强大的能力。很多有意思的问题都远远超出了经典正则表达式所能解决的范畴。下面来看一个例子，这个问题利用Perl模式很容易匹配，而任何经典正则表达式都无力解决（实际上这里是两个例子，一个是平衡方式，另一个是镜像方式）。

这本书本身是用pod编写的，这是Perl的一个轻量级文档系统。有关的更多内容请参见第23章的介绍，不过在某些方面，pod有些像其他发源于SGML的标记语言。它有标记和尖括号。与大多数其他标记语言类似，这些标记可以采用某种方式嵌套，正是由于这种嵌套方式，所以无法使用一个简单的老式正则表达式工具（如古老的*grep*）来搜索或处理这些标

记。这是因为，它们是嵌套的结构，这说明相应的搜索和处理工具也需要嵌套结构。而嵌套是标准的老式正则表达式无法做到的。幸运的是，Perl的正则表达式不是标准的，所以可以用来解析非传统的标记语言，如XM和HTML，还有pod。

Pod标记总是以一个大写字母开头，后面是一个或多个开始左尖括号。这部分很容易。难的是要查找最近的右尖括号（或大括号）来结束这个标记。对此有两种方法，取决于开始尖括号后面是否有空白符。

```
X<stuff>      # 平衡方式
X<< stuff >> # 镜像方式
```

如果没有空白符，就是在处理一个平衡标记，结束括号数必须与前面看到的开始括号数平衡，这也包括“stuff”内部的括号。

如果有空白符，标记就不是平衡标记，而是镜像标记：遇到空白符时，如果后面跟着的结束括号与开始括号同样多，则标记结束。在此中间，可以用任何符号表示尖括号（包括开始和结束尖括号），而且不要求它们嵌套，因为不会统计它们。

下面的例子让人望而生畏，这些也是这本书中实际用到的例子：

```
B<-0xR<HHH...>>
C<<< >>= >>>
C<0..(2Y<R<BITS>>)-1>
C<< BZ<><touch> SZ<><BZ<><-t> IZ<><time>> IZ<><file> >>
C<(?E<lt>!...)>
C<< !grep { !R<CODE>->($_) } keys R<HASH> >>
C<<< <HANDLE>, <<END >>>
C<(?<(R<COND>)R<IFTRUE>|R<IFFALSE>)>>
C<<< << R<EXPR> >>>
C<s/R<PATTERN>/R<REPLACEMENT>/>
I<Santa MarE<iacute>a>
X<<< < (left angle bracket);<< (left-shift) operator:@leftleft >>>
```

以下是相应文法的开始部分：

```
podtag::      capital either
capital::     uppercase_letter
either::      < balanced | mirrored >
```

这些会直接转换为子模式：

```
(?<podtag> (?&capital) (?&either) )
(?<capital> \p{uppercase_letter} )
(?<either> (?&balanced) | (?&mirrored) )
```

平衡的尖括号组就是由适当嵌套的尖括号包围的文本，如B<-0xR<HHHH>>。我们已经知道如何处理平衡模式，因为这与之前查找平衡小括号的问题很类似。

```
(?<balanced> < ( [^<>]++ | (?&balanced) )* > )
```



现在来看文法的最后一部分：镜像标记。镜像标记形如C<<< << R<EXPR> >>>。看到开始括号时必须查找同样多的结束括号，不过我们不用考虑统计中间的开始和结束括号（嗯，基本上可以这么说）。

```
(?<mirrored>    (?<open> <{2,}+ ) \s++
                  \s+
                  (?: (?&podtag) | \p{Any} ) *?
                  \s+
                  \s++ (??{ ">" x length $MATCH{open} })
)
```

<mirrored>的开始部分以占有方式捕获两个或更多地开始括号，把它们存储在<open>组中，以后需要统计时可以把它们用作反向引用。需要说明，我们只是把<open>用作一个命名捕获，而不是一个命名子规则，因为命名规则会隐藏其内部捕获。

中间部分抽取标记内容。这里我们必须当心，因为这些内容可能包含其他pod标记，而那些标记可能是平衡标记。不过，如果它只是一个镜像标记，则不会统计。我们使用了\p{Any}，这是Perl中(?s:.)的Unicode表示。换句话说，它会匹配任何字符，甚至可以匹配换行符<sup>注22</sup>。

最后一行内插这个表达式的结果，生成与第1行中看到的同样多的结束尖括号。

下面给出整个程序。

```
#!/usr/bin/env perl
# demo-podtags-core
use v5.14;
use strict;
use warnings;
use open qw(:std :utf8);          # 一个全UTF-8 workflow
use warnings FATAL => "utf8";      # 万一有输入编码错误
use re "/x";                       # 总希望模式易读
our %MATCH; *MATCH = \%;          # 别名%MATCH to %+以确保易读性

my $RX = qr{
    (? (DEFINE)
        (?<podtag>                (?&capital) (?&either)          )
        (?<capital>                \p{upper} )
        (?<either>                 (?&mirrored) | (?&balanced)      )
        (?<balanced>               < (?&contents) >                )
        (?<contents>               (?: (?&podtag) | (?&unangle) ) *  )
        (?<unangle>                [^<>]++                          )
        (?<mirrored>               (?<open> <{2,}+ ) \s++
                                    \s+
                                    (?: (?&podtag) | \p{Any} ) *?
                                    \s+
                                    \s++ (??{ ">" x length $MATCH{open} })
    )
}
```

---

注22：从技术上讲这并不正确。相对于Perl所能提供的一致性，在更高的层次上，\p{Any}还可以匹配特定于本地化环境的语言元素，如两字母连音和三字母连音。

```

    )
  )
};

@ARGV = glob("*.pod")          if @ARGV == 0 && -t STDIN;
die "usage: $0 [pods]\n"       if @ARGV == 0 && -t STDIN;

$/ = "";      # 段落模式，因为标记可能跨\n但不能跨\n\n
$| = 1;       # 更快地输出（为急躁的人着想）

while (<>) {
    while (/ (?<TAG> (?&podtag) ) $RX /g) {
        say $MATCH{TAG};
    }
}

```

还有几点要注意。要打印匹配，必须显式地将它保存到我们自己的作用域中的一个捕获组中（名为<TAG>的组）。当然，也可以使用\$&或\${^MATCH}或者诸如此类的组，不过关键是，如果在<podtag>调用中保存捕获组中的内容，这个调用返回时所保存的内容会丢失。

所以尽管可以使用这个技术来验证输入，不过要想取出你辛辛苦苦解析得到的结果，这种方法则可能有问题。虽然可以在代码中加入(?{ CODE }), 这样在执行过程中可以保存解析的部分结果，不过这种做法很快会变得很麻烦。更好的解决方法是使用下一节介绍的辅助模块。

## 文法

Damian Conway的Regexp::Grammars CPAN模块就是设计用来解决这个问题的，当然它还能解决更多其他问题。利用这个模块，我们不必像上一个例子中那样那么辛苦，用Perl写文法会变得很容易。这个模块确实是一个超棒的工具，有太多让人难以想象的特性，在这里无法深入解释，有关的详细信息请参见它的手册页。不过这里的简要介绍应该能吊起你的胃口来。

Regexp::Grammars实际上是一个文法编译器，与yacc有些相似。只不过不像yacc那样输出一个C程序，Regexp::Grammars会生成一个模式，你可以像使用其他模式一样使用这个模式。为做到这一点，它使用了一个技巧，这个技巧我们将在下一节讨论：它重载了qr//操作符，并将你提供给它的模式重写为一个不同的模式，这个模式将完成具体的工作。

下面我们重写前面的程序，这一次要使用Damian的这个模块。

```

#!/usr/bin/env perl
# demo-podtags-grammar
use v5.14;
use strict;
use warnings;
use open qw(:std :utf8);          # 一个全UTF-8工作流

```

```

use warnings FATAL => "utf8";          # 万一有输入编码错误
use re "/x";                          # 总希望模式易读

my $podtag = do { use Regexp::Grammars; qr{
    <podtag>
    <token: podtag>          <capital> <either>
    <token: capital>        \p{upper}
    <token: either>         <mirrored> | <balanced>
    <token: balanced>       \< <contents> \>
    <token: contents>       (?: <[podtag]> | <[unangle]> ) *
    <token: unangle>        [^<]+
    <token: mirrored>       <open=( \< {2,} )>
                             \s+
                             (?: <podtag> | \p{Any} ) *?
                             \s+
    </open>

    }xms;
};

@ARGV = glob("*.pod")      if @ARGV == 0 && -t STDIN;
die "usage: $0 [pods]\n"   if @ARGV == 0 && -t STDIN;

$/ = "";      # 段落模式，因为标记可能跨\n但不能跨\n\n
$| = 1;       # 更快地输出（为急躁的人着想）

while (<>) {
    while (/$podtag/g) {
        say $/{podtag}{capital},
            $/{podtag}{either}{" "};
    }
}

```

这个程序与前一个程序一样，可以解析同样的输入，文法看起来几乎是一样的。不过它在很多方面更胜一筹。它更容易写，而且我们认为它也更容易读。另外它还能做前一个版本做不到的事情。请看<mirrored>子例程。这里我们使用了Regexp::Grammars的一个特性，允许我们把开始左尖括号捕获到一个名为<open>的组中，然后只需写</open>就可以隐式地匹配相应数目的结束右尖括号。

也许更重要的是，我们的模式匹配和输出语句稍有些不同。匹配更简单，打印也更有意思。散列变量%/包含文法正则表达式成功匹配所创建的嵌套数据结构。

仅从这个打印你可能无法看出%/变量是结构化的。它与解析完全一致。假设你提供类似“Left C<B<nested>> an I<< N<inside>un>tag >> gets X<indexed> right”的程序输入。解析之后，我们使用Data::Dump模块来展示你将在%/中看到什么。

```

use Data::Dump; # 从CPAN
# 完成匹配，然后
dd \%/; # 将引用传入结果散列

```

你会得到以下输出：





可以看到，这个散列中实际上是运行解析时执行文法所生成的每一个产品的记录，包括嵌套标记。Regexp::Grammars比这里显示的要复杂得多。另外，一旦开始建立文法，如果你愿意，你的文法甚至可以相互共享正则表达式子例程。与上一节介绍的比较简单的文法相比，这里的共享要多得多（实际上，在前面比较简单的文法中根本没有共享）。如果你要完成复杂的解析，很有必要好好了解有关内容。

## 定义你自己的断言

你不能改变Perl的引擎如何工作（起码不能轻松改变：有关内容参见下一节），不过如果适当调整，确实可以改变引擎处理模式的方式。由于Perl对模式的解释与双引号字符串类似，所以可以利用重载字符串常量的强大功能，从而使你选择的文本序列自动地转换为另一个文本序列。

在下面的例子中，我们指定Perl遇到一个模式时会发生两个变换。首先我们定义了\tag，它在模式中出现时，会自动转换为(?:<.\*?>)，这会匹配大多数HTML和XML标记。其次，我们“重新定义”了\w元符号，使它只处理英语字母。

我们定义了一个包，名为Tagger，它隐藏了重载，不让我们的主程序看到。一旦完成上述工作，就可以写类似下面的代码：

```
use Tagger;
$_ = "<I>camel</I>";
say "Tagged camel found" if /\tag\w+\tag/;
```

下面给出Tagger.pm，这里采用Perl模块的形式给出（参见第11章）：

```
package Tagger;
use overload;

sub import { overload::constant "qr" => \&convert }

sub convert {
    my $re = shift;
    $re =~ s/ \\tag /<.*?>/xg;
    $re =~ s/ \\w /[A-Za-z]/xg;
    return $re;
}
1;
```

Tagger模块会在内插之前得到模式，所以如果避开内插，就可以避开重载，如下所示：

```
$re = '\tag\w+\tag'; # 这个字符串以\t（制表符）开头
print if /$re/;      # 匹配一个制表符，后面是一个"a"...
```

如果希望内插变量是定制的，则可以直接调用convert函数：

```
$re = '\tag\w+\tag'; #这个字符串以\t（制表符）开头
```

```
$re = Tagger::convert $re;    # 扩展\tag和\w
print if /$re/;               # $re 变成 <.*?>[A-Za-z]+<.*?>
```

现在如果你还不清楚Tagger模块中这些sub之类的东西是什么，很快你就会知道答案，因为这是第6章要讨论的内容。

## 其他候选引擎

从v5.10开始，甚至可以完全换掉Perl的正则表达式引擎，而替换为另外一个候选的模式匹配库。这样做的基本原理在`perlreapi`手册页中有详细说明。不过这个说明读起来很费劲，它只适合那些技艺高超的高级技术人员阅读。

不过你很幸运。多亏有了CPAN，不论你选择哪一个候选正则表达式引擎，可能都已经存在相应的Perl插件。使用这些插件时，你可以正常地编写你的模式，等到执行模式时，你选择的候选引擎就会接手执行。表5-18显示了更多CPAN模块，利用这些模块，可以在你的Perl代码中使用其他语言的正则表达式引擎（2011年秋季CPAN中就已经有这些候选引擎了）。等你读到这本书时，可能还会有更多模块出现，你可以再找找看。

表5-18：其他候选正则表达式引擎

模块	描述	版本	更新	当前维护人
<code>re::engine::LPEG</code>	LPEG正则表达式引擎	0.05	2010-07-09	François Perrad
<code>re::engine::RE2</code>	Russ Cox的RE2正则表达式引擎	0.08	2011-04-22	David Leadbeater
<code>re::engine::Plugin</code>	编写定制正则表达式引擎的通用API	0.09	2011-04-05	Vincent Pit
<code>re::engine::Plan9</code>	Plan9!的正则表达式	0.16	2010-03-31	Ævar Arnfjörð Bjarmason
<code>re::engine::Oniguruma</code>	Ruby的Oniguruma正则表达式引擎	0.05	2011-07-10	藤吾郎
<code>re::engine::Lua</code>	Lua的正则表达式引擎	0.06	2008-12-20	François Perrad
<code>re::engine::PCRE</code>	Phil Hazel的Perl兼容正则表达式引擎	0.17	2011-01-29	Ævar Arnfjörð Bjarmason

注意到这些作者有什么特殊的地方吗？其中超过三分之二的作者名字都无法用ASCII写出来。欢迎来到环球21世纪！

需要特别说明的一个引擎是Russ Cox的RE2库。这是一个C和C++库，用在Go编程语言



中（当然也用在其他地方）。有意思的是，它能非常好地维护Perl兼容性，包括很好的UTF-8支持，同时避免灾难性回溯的潜在问题。它这样做是有原因的，不同于Perl的引擎是一个递归回溯引擎，RE2使用了一种复合的NFA/DFA方法，对于有问题的情况不会过度深陷。

这对于时间敏感应用可能很重要，你可能想让用户提供他们自己的模式，不过不能让他们永无休止地搜索下去，你不能冒这个风险。这个引擎最早是为Google's Code Search编写的，时间对于这个应用极其宝贵。还可以通过RE2的Perl接口（<http://grep.cpan.me>）使用RE2。你可以在这个网站中输入一个搜索模式，它会在CPAN中运行所有模式。

一旦安装了`re::engine::RE2`<sup>注23</sup>，就使用很简单，如果你希望正则表达式使用RE2的引擎而不是内置的Perl引擎，只需要在这个正则表达式所在的词法作用域中增加一个`use re::engine::RE2`声明。就这么简单。

下面来看一个例子，从这里可以看到RE2远远胜过所有递归回溯引擎。首先，来测量运行常规Perl引擎的时间：

```
% time perl -E 'say (("a" x 17) =~ /a*a*a*a*a*a*a*a*a*a*[Bb]/i || 0)'
>/dev/null
1.564u 0.005s 0:01.57
% time perl -E 'say (("a" x 23) =~ /a*a*a*a*a*a*a*a*a*a*[Bb]/i || 0)'
>/dev/null
17.757u 0.025s 0:17.84
% time perl -E 'say (("a" x 29) =~ /a*a*a*a*a*a*a*a*a*a*[Bb]/i || 0)'
>/dev/null
127.965u 0.180s 2:09.20
```

再来测量使用RE2引擎的时间：

```
% time perl -Mre::engine::RE2 -E 'say (("a" x 500) =~
/a*a*a*a*a*a*a*a*a*a*[Bb]/i || 0)'
/dev/null
0.004u 0.002s 0:00.00
% time perl -Mre::engine::RE2 -E 'say (("a" x 5000) =~
/a*a*a*a*a*a*a*a*a*a*[Bb]/i || 0)'
/dev/null
0.004u 0.002s 0:00.00
% time perl -Mre::engine::RE2 -E 'say (("a" x 50000) =~
/a*a*a*a*a*a*a*a*a*a*[Bb]/i || 0)'
/dev/null
0.004u 0.002s 0:00.00
```

可以看到，利用RE2，运行时间不再与输入规模成正比增长，而只与正则表达式的规模有关。如果你的输入字符串像整个CPAN一样大，这就会很成问题。没错，这里演示的例子是人为设计的，不过存在这种问题的模式可能会非常频繁地出现。

---

注23：如果你还不知道如何安装，请参见第19章中的说明。

可以适当地配置`re::engine::RE2`，对它能处理的模式使用RE2，而它不能处理的模式则交给内置Perl引擎来处理，这就使它能够做到100%兼容。或者，如果你在提供一个外部服务，可以把它配置为只使用RE2，而不能使用内置Perl引擎作为退路，这样一来就不会遭遇拒绝服务的风险。

关于RE2的设计以及一般的无限自动机，更多内容请参见Russ Cox的系列文章（共3篇）：“Regular Expression Matching Can Be Simple and Fast”，“Regular Expression Matching: The Virtual Machine Approach”和“Regular Expression Matching in the Wild”。

# Unicode

如果你没听说过Unicode，说明你肯定在一个荒无人烟的孤岛上待了不止20年，除了一台手动打字机，身边别无它物。要知道，2010年初Unicode就已经庆祝过它的20岁生日了。即使你确实听说过它，也可能不太清楚它到底是什么，或者不知道如何使用。你不用为此觉得难堪，因为所有人都还在努力了解Unicode，甚至包括它的发明者也在不断探索中。尽管我们不指望这一章能全面涵盖Unicode的所有细节（甚至整本书都做不到），不过肯定能让你了解如何在Perl中使用Unicode。

如今，使用Unicode已经不再只是一种可有可无的选择，它已经成为我们不可缺少的东西。Web中的绝大多数资源都采用Unicode<sup>注1</sup>，很多大型资料库100%使用Unicode。因为Web浏览器总是尽其所能来适应Web服务器提供的字符集，你可能根本没有注意过现在Unicode的使用面有多广。未能提供可靠Unicode支持的编程语言已经远远落后于时代，而用这些语言编写的程序也与它们一同落伍。它们在20世纪80年代可能还很好，甚至到90年代也还不错，不过如今已经无法满足我们的需要。

## 为什么会这样呢？

计算机将字符存储为数字。刚开始的时候，这只是一一些小整数，只有5、6、7或8位长。EBCDIC使用8位，这是基于打孔卡片设计的。ASCII只用了8位字节中的7位，每个字节留出一位用作其他用途。实际上，这些用途实在太多，而且还相互矛盾。

所以，那时西方国家几乎所有人都会把字符与0~127（或0~255）范围内的小整数混淆起来。尽管这些字符比你的键盘上的按键多得多，但确实还不算很多，世界上不同地方的人们对于各个数字表示哪个字符都有自己的想法。

注1： 准确地讲，是Unicode的UTF-8编码。





值（例如，U+216B即“Ⅻ”的值为12）；打印宽度；双向性；分词和换行规则；以及字形变化。当然这还只是Unicode定义的一部分……

Perl最早在v5.6中开始尝试提供Unicode支持，不过直到v5.8我们才终于解决了一些重要的I/O问题。到了v5.12版本，其余的大多数问题都已经解决。利用v5.14版本的Perl（符合v6.0版本的Unicode标准），现在可以无缝地结合使用Unicode和Perl。嗯，也许还达不到百分之百的程度，但大多数情况下都能做到。起码比其他语言要强很多。

我们的意思是，Perl可以让容易的事情很容易完成，同时也不会妨碍处理那些困难的事情。首先要注意的一个“容易的事情”是：Perl允许字符串包含序数值任意大的字符。ASCII限制字符为7位序数值，Latin-1限制为8位序数值，而Unicode限制为21位序数值<sup>注2</sup>。不过Perl的字符没有限制为这么小的数。目前Perl的字符限制为64位（在64位机器上），尽管有限制，不过比Unicode本身提供的要多18,446,744,073,708,437,504个码点（嗯，之前我们说过序数值可以“任意大”，这么说确实不太严谨）。

在Unicode中，每个字符都有自己唯一的数，这称为它的码点（codepoint）。这正是Unicode得名的原因：所有不同字符的唯一通用的编码（unique, universal code）。例如，对于名为LATIN CAPITAL LETTER A的字符，相应的数是十进制数65，也就是十六进制数0x41。这通常写作U+0041；“U+”前缀是一个约定，表示这不是一个老数字，而是一个表示Unicode码点的数字。

如果你曾把“l”误以为是“1”，把“0”当成“o”，或者把“,”看成是“.”，应该很清楚人类很容易把这些字符弄混。你还知道计算机则绝对不会上当受骗。它不关心采用某种字体的字符看上去是什么样子。它关心的只是这个字符要做什么。例如，表6-2中所示的字符采用不同字体时看上去都差不多。

表6-2：容易混淆的大写字母A Unicode表示

字形	码点	类别	文字系统	名
A	U+00041	Lu	Latin	LATIN CAPITAL LETTER A
A	U+0FF21	Lu	Latin	FULLWIDTH LATIN CAPITAL LETTER A
A	U+00391	Lu	Greek	GREEK CAPITAL LETTER ALPHA
A	U+00410	Lu	Cyrillic	CYRILLIC CAPITAL LETTER A
A	U+1D5A0	Lu	Common	MATHEMATICAL SANS-SERIF CAPITAL A
A	U+1D670	Lu	Common	MATHEMATICAL MONOSPACE CAPITAL A

表中第一列是字符的字形。只有在当前词法作用域中声明了use utf8，你才能在你自己的代码中使用这样的Unicode直接量。大多数主流编辑器和窗口系统都提供了多种输入这些字符的方法，不过默认情况下可能并没有打开这些输入方法，所以需要先做些研究（如果

注2：严格地讲，只有20.087463位，因为Unicode只有0x110000个码点，而不是0x200000。



你不算太懒)。不过,即使你很懒也没关系:如果你不知道如何输一个字符,完全可以在别处找到这个字符,然后利用鼠标把它粘贴到你的代码中。

从外表来看,字形只是字符作用最小的一个方面,因为你无法知道采用别人的字体(而不是你自己的字体)时它会如何显示(甚至不能确定它能否显示)。可能在你看来某个字形还不错,不过不要太相信自己的眼睛,只能相信确凿的数字。第二列就是字符数值码的标准Unicode记法。在Perl中有很多描述码点的方法:

```
if (ord($somechar) == 0x0391) { .... }
$alpha = "\x{391}";
$alpha = "\N{U+391}";
$alpha = chr(0x391);
```

“码点”列后面是码点的两个最重要的属性:通用类别属性(列名为“类别”)和文字系统属性(列名为“文字系统”)。码点属性通常在模式中用作命名字符类, `\p{PROPERTY}` 匹配有该属性的所有码点, `\P{PROPERTY}` 匹配没有这个属性的码点。

```
/^\p{GC=Lu}+$/          # 所有大写字母
/^\p{script=Greek}+$/    # 希腊文字
/[\P{script=Latin}\P{script=Common}]/ # 不是交集
```

如果字符串中包含非Latin或非Common的码点,最后这个模式就很适用。因为属性名中不考虑大小写、空白符和下画线,所以可以用你喜欢的任何方式指定格式。如果你想更可读一些,写为全小写的 `\p{gc=modifier_letter}` 和全大写的 `\P{SC=INHERITED}`,这是完全可以的:Perl并不关心。或者如果你愿意,也可以反过来。

除了以上这种两部分属性,Perl还为所有通用类别和文字系统提供了一个单元素别名,所以如果你愿意,可以直接使用 `\p{Lu}` 和 `\p{Greek}`。例如,如果想确保字符串中只包含拉丁语和希腊语字符,可以写为:

```
$mylang = qr/[\p{Latin}\p{Greek}\p{Common}\p{Inherited}]/;
if ($string =~ /\A$mylang+\z/) { ... }
```

我们增加了一个Common文字系统,对应多个文字系统共同的字符,如数字和标点符号,另外还增加了Inherited,这表示组合标记(通常是变音符)等符号,这些符号的文字系统就是关联了这些符号的基本码点的文字系统。连接码点与ASCII中的码点完全不同,所以熟悉ASCII的人刚开始接触Unicode时最先困惑的就是连接码点。在ASCII中与它最接近的概念可能是使用退格键加粗,只不过在Unicode中连接码点会自动应用到前一个码点,所以没有必要使用退格键。这些内容将在下面的“字形和规范化”一节中讨论。

你会注意到,这一章中我们开始区别“码点”和“字符”。在其他地方(包括这本书中的其他章节),通常交替使用这两个词,有时字符也用来表示“字形”,不过在这里我们需要更明确一些。在看上去像一个整数列表的字符串中,码点特指构成这个字符串的单个整



数，而字符概念有些模糊，可能表示码点，也可能表示人们看到的字形。更一般地，要注意人们谈到“字符”时可能是指3种甚至4种不同的东西。

表中最后一列是字符名。嗯，实际上应该是码点名。要在你的代码中指定码点名，首先要用charnames模块加载码点，然后可以使用\N{...}来引用，如下：

```
use charnames qw(:full);

$alpha      = "\N{GREEK CAPITAL LETTER ALPHA}";
$alpha_code = ord "\N{GREEK CAPITAL LETTER ALPHA}";
if ($string =~ /\N{GREEK CAPITAL LETTER ALPHA}/) { ... }
```

用名字指定码点比用数字要好得多。这样可以让你的代码更容易理解。

利用\N{...}还能做很多其他事情，有关内容参见第29章中的“charnames”一节。

## 展示，不要告诉

一幅图强过千言万语，如此说来，在程序中放入你想要的具体字符肯定很有意义。你可能想告诉Perl你的源代码是Unicode字符，而不只是简单的字节<sup>注3</sup>。并不要求一定这样做，不过如果可以在源代码中输入真正的Unicode，有些问题会容易一些。

到目前为止，Perl假设源代码中的所有字符都是ASCII字符，除非你明确告诉它不是这样（不过，有可能将来某一天Perl会默认地认为源代码都是Unicode字符）。完全可以用前面提到的迂回方法指定Unicode码点，不过采用这种做法时，直接量将处理为单独的字节。如果Perl看到一个直接量UTF-8字符，则它不会意识到应当把它当作一个逻辑字符，而会显示为1个、2个、3个甚至4个单独的Perl字符，各个字符的序数值都小于256。你不希望这样，所以要使用以下声明：

```
use v5.14;          # 包含unicode_strings特性
use utf8;           # 处理UTF-8直接量
```

前一个声明确保序数值在128~255的码点要处理为Unicode字符串，而第二个声明告诉Perl编译器整个源文件都采用Unicode的UTF-8编码。由于有utf8 pragma，现在你可以在字符串和正则表达式直接量中使用Unicode了。

```
my $dwarf      = "þórinð Eikinskjalði";
my $search     = "búsqueda";
my $measure    = "Ångström";
my $show       = "à contre-cœur";
my $motto      = "👯❤️👉";
```

这样读起来就容易多了，不过输入可能没有书写那么容易：

---

注3： 你可能想把它们称为“8位字节”（octet）。没问题，不过我们认为如今这两个词基本上是同义词，所以我们还是用比较通俗的说法。

```

use charnames qw(:full);

my $dwarf    = "\N{LATIN CAPITAL LETTER THORN}\N{LATIN SMALL LETTER
               O WITH ACUTE}rinn Eikinskjalldi";
my $search   = "b\N{LATIN SMALL LETTER U WITH ACUTE}squeda";
my $measure  = "A\N{COMBINING RING ABOVE}ngstro\N{COMBINING DIAERESIS}m";
my $show     = "\N{LATIN SMALL LETTER A WITH GRAVE} contre-c\N{LATIN
               SMALL LIGATURE OE}ur";
my $motto    = "\N{FAMILY} \N{GROWING HEART} \N{DROMEDARY CAMEL}";

```

所以，更好的办法是在代码中放入秘密魔法数，如下：

```

my $dwarf    = "\x{DE}\x{F3}rinn Eikinskjalldi";

my $search   = "b\x{FA}squeda";
my $measure  = "\x{C5}ngstr\x{F6}m";
my $show     = "\x{E0} contre-c\x{153}ur";
my $motto    = "\x{1F46A} \x{1F497} \x{1F42A}";

```

不过，这还不是全部。基于utf8声明，现在还可以在Perl标识符中使用Unicode。

```

# 一些字符集
my @Is0      = qw( Latin1 Latin2 Latin15 );
my @μsoft    = qw( cp852 cp1251 cp1252 );
my @鲤       = qw( koi8-f koi8-u koi8-r );

# 是否包含无结果返回的答案
my $INCLUIR_NINGUNOS = 0;

# 是否考虑变音字符的问题
my $SI_IMPORTAN_MARCAS_DIACRÍTICAS = 0;

# 把<< 看做是"直到型"操作符 :)
my @ciudades_españolas = ordenar_a_la_española(<<'LA_ÚLTIMA' =~ /\S.*\S/g);
...
...
LA_ÚLTIMA

my $déjà_imprimée; # le nom d'une ville

# 希腊文hypermegas
my @ὐνέπμεγας = ( );

# 正确，现在来展示
my $indino = umopəpɪsɔn($input);

```

在实际中，如果你在标记符中使用了非英语的名字，可能希望在注释里用相应的语言做出说明。这样一来，全世界的人就能更容易地用他们自己的语言来写Perl代码，而不是要求他们都学英语。

当前要限制Unicode标识符只能作为私有变量。这是由于全局变量特定的存储方式，另外也因为模块名会以特定的方式映射到本地文件系统。对于这两个限制，可能会在不远的将来取消第一个限制，不过第二个限制还需要好好研究。

## 获取Unicode数据

在内部，Perl保持所有码点都采用一种与Unicode兼容的格式，这意味着下面21位与Unicode相同，就像Unicode的下面8位与Latin-1相同一样。这些码点在内部具体如何存储无需普通Perl用户操心。不过，一旦要与外部世界交流，你就必须与所提供的输入数据交互，并以适当的格式生成数据，让接收数据的程序能够接受。Perl中的字符已经由其外部格式解码为抽象字符，不过要发出这些字符，必须再把它们编码为你期望的某种格式。如果忘记这样做，则通常会成一些警告，指示有“宽字符”或者“格式不正确的UTF-8字符”。

要标记整个流的编码，Perl主要提供了两种方法，另外还有一些快捷方式来提供方便。如果流已经打开，则可以为binmode函数传入第二个参数来设置流编码：

```
binmode(STDIN, ":encoding(CP1252)")
|| die "can't binmode to cp1252: $!";
binmode(STDOUT, ":encoding(UTF-8)")
|| die "can't binmode to UTF-8: $!";
```

如果还没有打开文件，可以在打开文件的open调用中使用模式参数指定编码。

```
open(OUTPUT, "> :raw :encoding(UTF-16LE) :crlf", $filename)
    or die "can't open $filename: $!";
print OUTPUT for @stuff;
close(OUTPUT) or die "couldn't close $filename: $!";
```

在输入中，:crlf层将\n转换为\r\n，在输出时，则正好相反，将把\r\n转换为\n。在Windows系统上，如果打开文件处于“文本”模式，则会默认地启用这一层，不过在UNIX上如果想要这种行为，则需要明确指定。有关I/O层的更多内容参见PerlIO手册页。

不过Unicode并非只有\n和\r\n行终止符。目前Unicode可以识别8个不同的换行序列，包括7个码点和一个两码点\r\n序列：

```
U+000A LINE FEED (LF)
U+000B LINE TABULATION
U+000C FORM FEED (FF)
U+000D CARRIAGE RETURN (CR)
U+0085 NEXT LINE (NEL)
U+2028 LINE SEPARATOR
U+2029 PARAGRAPH SEPARATOR
```

通常并没有特殊的层来处理Unicode换行序列，不过如果你能把整个文件都读入内存，那么可以很容易地把所有换行序列都转换为换行符：

```
$complete_file =~ s/\R/\n/g;
```

或者把它们分解为一个行列表，列表中各行都不含行终止符：

```
@lines = split(/\R/, $complete_file);
```



可以使用`open pragma`为所有新打开的文件句柄设置编码。例如，对于没有指定编码的`open`调用，如果要指定其默认编码为UTF-8（就像STDIN、STDOUT和STDERR一样），可以使用如下代码：

```
use open qw( :encoding(UTF-8) :std );
```

如果想把标准流设置为UTF-8而不是二进制，可以在每次执行时使用`-CS`命令行选项，或者将`PERL_UNICODE`环境变量设置为“S”。如果将它设置为“D”，那么打开时所有未指定编码层的句柄都默认为UTF-8文本而不是二进制字节数据。参见第17章。

如果使用了`-C`命令行选项或`PERL_UNICODE`环境变量，那么甚至处理二进制流的UNIX程序也需要调用`binmode`，而通常只有Windows用户（或者编写可移植程序的人）才需要这样做。这会破坏原来的一些UNIX程序，因为这些程序认为如果没有特别说明，它们得到的是二进制数据而不是文本。不过如果原来的程序不知道如何解码UTF-8文本，则不会受此影响。

按从高到低排序（较高的设置会覆盖下面较低的设置），设置流编码的各种机制的优先级如下：

1. 对已经打开的文件句柄调用`binmode`。
2. 如果`open`调用有3个或更多参数，在第二个参数中包含一个编码层。
3. 使用`open pragma`。
4. 使用`-C`命令行开关。
5. 使用`PERL_UNICODE`环境变量。

对此有一个例外：`DATA`句柄。`use utf8`或`open pragma`都不会影响这个句柄，所以你仍然要自己设置它的编码：

```
binmode(DATA, ":encoding(UTF-8)");
```

由于`utf8`和UTF-8编码层的实现方式，它们通常不会对格式不合法的输入抛出异常。要想抛出异常，可以在代码中增加以下声明：

```
use warnings FATAL => "utf8";
```

在v5.14中，Perl有3个子警告，它们都属于“`utf8`”警告组，有时你可能希望区别这3个子警告。它们分别是：

#### *nonchar*

Unicode留出了66个码点作为非字符码点。这些码点的通用类别为Unassigned (Cn)，不会为它们赋值。这些码点不能用在开放信息交换中，所以代码可以把这些码点作为内部卫哨混杂在字符数据中，总能将它们与字符数据区分开。这些非字符码点包

括U+FDD0到U+FDEF（32个码点），以及各个平面的最后两个码点（34个码点，因此其十六进制码以FFFE或FFFF结尾）。有些情况下，你可能希望允许使用这些码点，例如在两个合作进程之间使用，它们可以将这些码点用作共享卫哨。如果是这样，需要有以下声明：

```
no warnings "nonchar";
```

#### *surrogate*

这些码点是为UTF-16保留的。实际上没有理由启用这些码点，符合UTF-16的进程无法交换这些码点，因为UTF-16代理不能处理（不过UTF-8和UTF-32代理可以，所以我们更倾向于UTF-8和UTF-32）。

#### *non\_unicode*

最大的Unicode码点是U+10FFFF，不过Perl知道如何接受更大的码点，甚至可以达到平台所允许的最大无符号数。取决于不同的设置和具体环境，如果试图处理或输出大于正常范围的码点，Perl可能会发出警告（使用警告类别“non\_unicode”，这是“utf8”的一个子类别）。例如，“uc(0x11\_0000)”会生成这个警告，并返回输入参数作为结果，因为每个非Unicode码点的大写就是码点本身。

在这些码点中，到目前为止非Unicode码点是最有用的，你可能希望在内部使用中允许使用这些码点。

```
no warnings "non_unicode";
```

以下是一种可能的用法。对于ASCII可以有：

```
tr[\x00-\x7F][\x80-\xFF];
```

对于Unicode可以有：

```
tr[\x{00_0000}-\x{10_FFFF}][\x{20_0000}-\x{30_FFFF}];
```

也就是说，它们将所有码点从各自字符集的合法范围重新映射到一个非法范围。为什么要这样做？这样做是为了标记某些文本，表示你不想匹配这些文本。不过等你完成工作，一定要把它们重新映射回原来的合法范围。

## Encode模块

标准Encode模块通常都是隐式使用，而不是显式使用。只要向bin mode或open传入一个:encoding(ENC)参数，就会自动加载这个模块。

不过，有时你会发现一些编码数据来自某个流，但这个流的编码并不是由你设置的，所以在处理这些数据之前，你必须手动地对其解码。这些编码字符串可能来自你的程序以外的其他地方，如一个环境变量、程序参数、CGI参数或一个数据库字段。嗯，甚至可能会看

到这样一些“文本”文件，其中有些行采用一种编码，而另外一些行采用不同的编码。你肯定会看到乱码（mojibake）。

在这些情况下，需要依靠Encode模块更显式地管理编码和解码。最常使用的函数就是（毫不奇怪）encode和decode。如果有一些原始的外部数据采用某种编码形式存储为字节，可以调用decode将它们转换为抽象内部字符。另一方面，如果你有一些抽象内部字符，希望把它们转换为某种特定的编码机制，则要调用encode。

```
use Encode qw(encode decode);
$chars = decode("shiftjis", $bytes);
$bytes = encode("MIME-Header-ISO_2022_JP", $chars);
```

例如，如果你很肯定地知道最终编码设置为UTF-8，可以如下对@ARGV解码：

```
# 这类似于perl -CA
if (grep /\P{ASCII}/ => @ARGV) {
    @ARGV = map { decode("UTF-8", $_) } @ARGV;
}
```

对于不是使用全UTF-8环境的人来说，假设最终编码总是UTF-8可不是一个好主意。最终编码可能是一个本地化环境编码。尽管标准Encode模块不支持对本地化环境敏感的操作，但CPAN Encode::Locale模块支持。可以如下使用：

```
use Encode;
use Encode::Locale;

# 使用"locale"作为参数提供给encode/decode
@ARGV = map { decode(locale => $_) } @ARGV;

# 或者作为一个流提供给binmode或open
binmode $some_fh, ":encoding(locale)";

binmode STDIN, ":encoding(console_in)" if -t STDIN;
binmode STDOUT, ":encoding(console_out)" if -t STDOUT;
binmode STDERR, ":encoding(console_out)" if -t STDERR;
```

数据库中往往需要处理手动编码和解码。这取决于具体的数据库系统。基于简单的DBM文件，底层库希望得到字节而不是码点字符串，所以不能在DBM文件中直接使用常规Unicode文本。如果试图这么做，你会得到一个子例程异常，指出有一个“宽字符”（Wide character）。要在一个%dbhash DBM散列中存储一个Unicode键-数据对，首先要将它们编码为UTF-8：

```
use Encode qw(encode decode);

# 假设$uni_key和$uni_value是抽象Unicode字符串

$enc_key = encode("UTF-8", $uni_key);
$enc_value = encode("UTF-8", $uni_value);
$dbhash{$enc_key} = $enc_value;
```



与之对应的动作是获取一个Unicode值，因此在使用之前需要先对键编码，然后获取值，得到值之后再返回解码的值：

```
use DB_File;
use Encode qw(encode decode);

tie %dbhash, "DB_File", "pathname";

# $uni_key包含一个正常的Perl字符串（抽象Unicode字符串）
$enc_key = encode("UTF-8", $uni_key);
$enc_value = $dbhash{$enc_key};
$uni_value = decode("UTF-8", $enc_value);
```

现在可以像处理其他Perl字符串一样处理返回的\$uni\_value。之前，它只包含字节，只是以字符串形式存储的小于256的整数（这些整数并不是Unicode码点）。

从v5.8.4开始，你还可以使用标准DBM\_Filter模块，它会为你透明地处理编码和解码。

```
use DB_File;
use DBM_Filter;

use Encode qw(encode decode);

$dbobj = tie %dbhash, "DB_File", "pathname";
$dbobj->Filter_Value("utf8");

# $uni_key 包含一个正常的Perl字符串（抽象Unicode）
$uni_value = $dbhash{$uni_key};
```

## 张冠李戴

如果你只了解ASCII，可能对文本中的大小写有一些假设，不过这些假设在Unicode中几乎都是不正确的。在ASCII中只有大写字母和小写字母，而在Unicode中还有第三种大小写形式，称为标题形式（titlecase）。这在英语中用的并不多，不过在源自拉丁语或希腊语的其他文字系统中确实经常出现。

通常标题形式就等同于大写，不过并不总是这样。只有当首字母需要大写而其余部分不大写的情况下才会使用标题形式。有些Unicode码点看上去就像是两个字母并排打印，不过它们实际上是一个码点。如果要求一个单词仅第一部分首字母大写而其余不大写，对这个单词使用标题形式时，只会将适当的部分首字母大写。这种用法主要是为了支持遗留的编码，而如今更常见的做法是使用码点，其标题形式映射将生成两个单独的码点，包括一个大写和一个小写。以下是一个遗留字符的例子：

```
use v5.14;
use charnames qw(:full);
my $beast = "\N{LATIN SMALL LETTER DZ}ur";
say for $beast, ucfirst($beast), uc($beast);
```

这会打印“dzur”、“Dzur”和“DZUR”，它们都只有3个码点长。

有些字母没有大小写，而有些非字母确实有大小写。字母形式（Lettercase）在文字系统世界里相当少见。Unicode v6.0支持近100个文字系统，其中只有8个区分字符的大小写：Armenian、Coptic、Cyrillic、Deseret、Georgian、Glagolitic、Greek和Latin文字系统，另外Common和Inherited文字系统的字符有大小写。所有其他文字系统都不区分大小写。

如果存在大小写映射，字符串的长度可能会改变。基于简单的大小写映射，一个字符串的大小写映射总是与原字符串长度相同，不过如果使用完全大小写映射，可能不一定如此。例如，“tschüß”的大写是“TSCHÜSS”，长度会多一个字符。

采用某种大小写的不同字符串映射到另一个大小写形式时，可能会映射为相同的字符串。例如，小写希腊字符“σ”和“ς”有相同的大写形式“Σ”，而这只是一个简单的例子。为了正常（或者起码不那么不正常）地处理所有这些变化情况，对于不区分大小写的比较，还需要第4种大小写映射，称为foldcase。根据定义，有相同foldcase的字符串在不区分大小写的情况下完全相等。

Perl总是使用/i模式修饰符来支持大小写无关的匹配，这就是在比较其casefold。从v5.16开始可以直接支持fc函数，允许比较两个字符串的foldcase形式来确定它们是否是相同字符串的不同大小写形式。在v5.16之前，可以从Unicode::CaseFold CPAN模块得到fc函数。

检查你的perlfunc手册页以及perldelta中的发行说明，来了解你的版本中是否已经提供这个特性。如果有，可能会有一个内插转换转义\F，它的做法类似于\L和\U，不过会转换为foldcase形式。

使用非ASCII字符时还有一点让人意外，大小写映射不一定是可逆的操作。例如，lc("ß")是"ß"，而uc("ß")是“SS”，lc("SS")是“ss”，这并不是最开始的那个字符(" ")。不能保证经过一圈大小写映射又回到最初的那个字符，不只是这种特殊的两字符组合有这个特点。还记得吧，希腊字母“σ”是原形，不过用在单词末尾时要写为“ς”，它们有相同的大写形式“Σ”。这个字母经过一轮大小写映射后(lc(uc("ς")))，即来回映射后，并不会还原到最开始的“ς”，而只会得到“σ”。

对于区分大小写的字符，并不是所有字符在大小写映射时都会改变。在Unicode中，如果说一个字符是小写，这并不意味着它一定有一个大写形式或标题形式的大小写映射。例如，uc("M<sup>c</sup>Kinley")是“M<sup>c</sup>KINLEY”，因为MODIFIER LETTER SMALL C是小写，但是大小写映射时它并不改变，这看起来不太对劲。类似的，小体大写字母实际上是小写字母，因为它们低于字体的x-height。在“BOULDER CAMERA”中，每个单词的第一个字母是大写，其余的均为小写。

甚至被认为是小写的字符并不一定都是字母。大小写属性与通用类别属性不同。罗马数字



是区分大小写的数字，例如，“VIII”和“viii”分别是数字8的大写和小写。甚至还有一些被认为是小写的字母的通用类别属性是GC=Lm而不是GC=Ll。

要把一个单词置为“标题”形式，一般的ASCII策略是使用ucfirst(lc(\$s))，而这种做法在Unicode中不能保证一定正确，因为把小写版本转换为标题形式并不完全等同于将原字符转换为标题形式。对于其他组合也是如此。正确的方法是首先将首字母变为标题形式，然后将其余字符变为小写，为此可以显式调用函数，或者利用一个正则表达式替换来完成：

```
$tc = ucfirst(substr($s, 0, 1)) . lc(substr($s, 1));  
  
s/(\w)(\w*)/\u$1\L$2/g;
```

除了通用类别，Unicode还有很多其他与大小写有关的属性。表6-3给出了Unicode v6.0中支持的一些属性。它们都是二进制属性，所以如果你愿意，可以直接使用它们的单元素形式。可以不写为\p{CWCM=Yes}和\p{CWCM=No}，而是写为\p{CWCM}表示所有包含该属性的码点，另外用\P{CWCM}表示所有不包含这个属性的码点。

表6-3：与大小写有关的属性

短格式	长格式
Cased	Cased
Lower	Lowercase
Title	Titlecase
Upper	Uppercase
CWL	Changes_When_Lowercased
CWT	Changes_When_Titlecased
CWU	Changes_When_Uppercased
CWCM	Changes_When_Casemapped
CWCF	Changes_When_Casefolded
CWKCF	Changes_When_NFKC_Casefolded
CI	Case_Ignorable
SD	Soft_Dotted

Lower和Upper属性将匹配有相应属性的所有码点，而不只是字母。目前还没有非字母的标题形式码点，所以（现在）Title等同于gc=Lt。不过，如果有/i，这三者都与Cased属性相同，而不是特定于字母。不区分大小写地使用gc=Lt只等同于Case\_Letter。

## 字形和规范化

我们已经提到，类似LATIN SMALL LETTER DZ的字符占一个码点，但是对最终用户来说这看起



来可能像两个字符。相反的情况也存在，而且更为常见。也就是说，一个用户可见的字符（一个字形）可能需要多个码点来表示。考虑有一个或多个变音符的字母，如“résumé”中的两个é。它们可能分别有一个码点，也可能有两个。甚至有可能其中一个é是一个字母码点，而另外一个字母后面跟一个组合标记。根据设计，如果只从外形上来看，你无法做出区分，因为按规范认为它们是等价的。这会对几乎所有文本处理带来重要的影响，而这与我们之前所说的字形并不重要的说法完全对立。从这个意义上讲字形是最重要的。

可以用组合字符将“n”改为“ñ”，将“c”改为“ç”，将“o”改为“ô”，或者将“u”改为“û”。前3个变换需要一个组合标记，而最后一个需要两个组合标记。实际上，对于这些组合标记的使用没有任何限制。只要你愿意，可以一直堆叠这些组合标记，还可以创建人们以前从来没有见过的字符。

所有这些问题要求我们认真地重新考虑和重新编写所有软件。可以考虑一下字体系统要做的调整（不要因为困难干脆放弃，这可不是正确的选择）。处理文本的程序可能需要大改。即使概念上很简单的程序（例如只是逆置一个字符串）也可能出问题，因为如果你是按码点而不是按字形逆置，就会把组合字符从一个基字符移到另一个基字符上。Niño、María和François会变成ñniN、áiraM和siQcnarF。

考虑这样一个字形：它包括一个基本字母码点A，后面跟有两个组合标记，分别称为X和Y。这些标记应用的顺序会有影响吗？AXY和AYX一样吗？有时它们是一样的，但有时它们并不相同。对于像“ō”的字形，这个顺序没有影响，因为一个标记在上，另一个标记在下。由于顺序并不重要，所以如果一个字形显示为一个LATIN LETTER SMALL O，后面是一个COMBINING OGONEK，然后是一个COMBINING MACRON，而另一个字形的组合标记顺序正好相反（即先是一个LATIN LETTER SMALL O，后面是一个COMBINING MACRON，然后是一个COMBINING OGONEK），程序对这两个字形的处理将完全相同。不过，如果一个字形的两个组合标记都在字母的同一部分，那么它们应用的顺序就会有影响了。稍后介绍有关的更多内容。

还有更困难的。Unicode中有些预作字符，允许来回转换，即可以从遗留字符集转换到Unicode，然后再转换回来。拉丁文有大约500个这样的字符，希腊文有大约250个这样的字符。除此之外还有更多。

例如，“é”可能是码点U+00E9，LATIN LETTER SMALL E WITH ACUTE。这只是一个码点。不过这里有一点要注意：处理这个码点时要把它看作是一个两码点的字形（就好像这个字形显示为一个LATIN LETTER SMALL E，而且后面跟有一个COMBINING ACUTE ACCENT）。

对于逻辑上包含多个标记的字形，甚至可以有更多变形，因为其中一些字形可能以某个预作字符开始，而这个预作字符本身已经有一个标记，然后再加另外一个字符。

为了帮助解决所有这些问题，Unicode有一个明确的过程，称为规范化（normalization）。

按照Unicode术语表 (<http://unicode.org/glossary/>) 中的定义, 规范化是指“从文本数据中删除等价序列的候选表示, 将数据转换为一种可以完成二进制等价性比较的形式”。换句话说, 它为每一个有需要的语义实体提供一个唯一的标识, 所以所有一对多映射就不复存在了。

以下给出4种Unicode规范化形式(范式):

- 范式D (NFD)由规范化分解构成。
- 范式C (NFC)由规范化分解和规范化组合构成。
- 范式KD (NFKD) 由兼容性分解构成。
- 范式KC (NFKC)由兼容性分解和规范化组合构成。

正常情况下, 你可能希望使用规范化形式, 因为规范化为兼容性形式会损失信息。例如, `NFKD("™")` 会返回常规的两字符的字符串“TM”。对于搜索以及与搜索有关的应用, 这可能是你想要的, 不过对于大多数应用来说, 规范化分解都要优于兼容性分解。

除非你自己完成规范化, 否则字符串在程序中不一定用NFD或NFC显示; 有些字符串没有采用这两种范式。考虑一个字符“ō”, 这就是一个小体拉丁字母“o”, 再在它上面加一个波浪线和一个长音符号(而不是一个长音符号和波浪线)。取决于具体的规范化形式, 这个字形可能占1到3个码点: 如果是NFC则为“`\x{22D}`”, 如果是NFD则为“`\x{6F}\x{303}\x{304}`”, 或者如果既不是NFC又不是NFD则为“`\x{F5}\x{304}`”。表6-4显示了小体拉丁字母“o”加一个波浪线(可能还有一个长音符号)的7种变形。

表6-4: 规范化表示

N	字形	NFC?	NFD?	直接量	码点
1	ō	✓	—	" <code>\x{F5}</code> "	LATIN SMALL LETTER O WITH TILDE
2	ō	—	✓	" <code>o\x{303}</code> "	LATIN SMALL LETTER O, COMBINING TILDE
3	ō	✓	—	" <code>\x{22D}</code> "	LATIN SMALL LETTER O WITH TILDE AND MACRON
4	ō	—	—	" <code>\x{F5}\x{304}</code> "	LATIN SMALL LETTER O WITH TILDE, COMBINING MACRON
5	ō	—	✓	" <code>o\x{303}\x{304}</code> "	LATIN SMALL LETTER O, COMBINING TILDE, COMBINING MACRON
6	ō	—	✓	" <code>o\x{304}\x{303}</code> "	LATIN SMALL LETTER O, COMBINING MACRON, COMBINING TILDE
7	ō	✓	—	" <code>\x{14D}\x{303}</code> "	LATIN SMALL LETTER O WITH MACRON, COMBINING TILDE



Perl中，标准Unicode::Normalize模块会为你处理规范化函数。对此有一个很好的技巧：首先要通过NFD规范化所有Unicode输入，而最后要做的是通过NFC规范化所有Unicode输出。换句话说，可以如下所示：

```
use v5.14;
use strict;
use warnings;
use warnings FATAL => "utf8"; # 抛出编码错误异常
use open qw(:std :utf8);

use Unicode::Normalize qw(NFD NFC);

while (my $line = <>) {
    $line = NFD($line);
    ...
} continue {
    print NFC($line);
}
```

这会读入UTF-8输入，并自动解码，如果存在格式不合法的UTF-8，会抛出一个异常。在循环中，首先它将输入字符串规范化为其规范化分解形式。换句话说，完全分解所有预作字符（这会让简化论者很满意）。它还会重排关联到基本码点<sup>注4</sup>上不同位置的所有标记，使它们有一个可靠的顺序。

除非完成规范化，否则甚至无法开始处理组合字符问题。考虑表6-4所示的不同字形：

- 第4个既不是NFC也不是NFD。所以会出现这些问题。
- 假设强制NFD，1会变成2，3和4会变成5，7会变成6。
- 假设强制NFC，2会变成1，4和5变成3，6会变成7。
- 这说明，通过规范化为NFD或NFC，可以完成一个简单的eq来分别测试1~2，3~5和6~7的相等性。
- 但要注意这是3个不同的集合。

不过有一个好消息，Perl模式对字形提供了很好的支持（如果你知道如何使用的话）。正则表达式中的\X能匹配一个用户可见的字符，按Unicode的说法，这称为一个字形簇<sup>注5</sup>。

并不是所有字形簇都是一个基本码点加上0个或多个组合码点，不过大多数都是如此。“\r\n”是一个相当常用的两字符字形，它就没有任何组合字符，这个字形通常称为CRLF。\\X将CRLF匹配为一个字形簇，因为这只是一个用户可见字符。日语中也有两个不

---

注4： 有意思的是，它们要根据其规范化组合类来重排顺序。

注5： 按照非常技术性的说法，Perl的\\X匹配Unicode标准中所称的扩展字形簇。编写标准的人显然被这个词收买了。



包含组合字符的扩展字形：HALFWIDTH KATAKANA VOICED SOUND MARK和HALFWIDTH KATAKANA SEMI-VOICED SOUND MARK。

不过，大多数情况下，都可以把字形簇看作是一个基本字符(`\p{Grapheme_Base}`)再加上一定数量的组合字符变形选择器、日语语音标记或0宽度连接或非连接字符(`\p{Grapheme_Extend}`\*)紧跟在它后面，但CRLF对有所例外。

实际上，可以把字形簇就认为是一个字形<sup>注6</sup>。

Perl模式中的`\X`可以不加区别地匹配以上7种情况，甚至不要求采用规范化形式。这一点没错，不过然后呢？从现在开始就不那么容易了。因为除了知道字形是字形外，如果你想对字形有更多的了解，就必须对模式匹配相当精通。要完成以下匹配，前提是要有NFD：

- `/^o/`报告所有这7个变形都以一个“o”开头。
- `/^o\x{COMBINING TILDE}/`报告1–5以一个“o”和一个波浪线开头，但6和7不是。
- `/^o\pM*?\x{COMBINING TILDE}/`才能匹配所有这7个字符。

这只是匹配一个完整字符所做的尝试，不过还有很多问题没有解决，例如是否使用`\p{Grapheme_Extend}`而不是`\pM`，另外是否使用`\p{Grapheme_Base}`（如果有）取代`\PM`：

```
$o_tilde_rx = qr{ o \pM *? \x{COMBINING TILDE} \pM* }x;
```

要完成不区分重音符的字符串比较，可以参见下一节“Unicode文本比较和排序”，那里将提供一种更容易的方法。

Perl内核中唯一了解字形的只有模式中的`\X`。类似`substr`、`length`、`index`、`rindex`和`pos`等内置函数只会在码点级访问字符串，即粒度是码点而不是字形。所以`\X`相当于一把锤子，所有Unicode就像是钉子。确实有很多很多的钉子。

假设要逐码点地逆置“*crème brûlée*”。如果规范化为NFD，最后会得到“*éélurb emèrc*”，而不是你想要的“*eélûrb emèrc*”。如果使用`\X`抽出一组字形，然后按字形逆置。

```
use v5.14;
use utf8;
my $cb = "crème brûlée";
my $bc = join("", => reverse($cb =~ /\X/g));
say $bc; # "eélûrb emèrc"
```

假设`$cb`是“*crème brûlée*”，下面对处理码点和处理地形的的方式做个比较：

```
my $char_length = length($cb); # 15 或 12
my $graph_count = 0;
```

---

注6： 我们就是这样做的。我们可没有被“字形簇”这个词收买。

```
$graph_count++ while $cb =~ /\X/g; # 12
```

可以如下取出第一部分：

```
my $piece = substr($cb, 0, 5);      # "crèm" 或 "crème"  
my($piece) = $cb =~ /\X{5}/;      # "crème"
```

采用以下方式改变后一部分：

```
substr($cb, -6) = "fraîche";      # "crème brfraîche"或"crème fraîche"  
$cb =~ s/\X{6}$fraîche/;      # "crème brûlée"
```

下面会插入“bien”：

```
substr($cb, 5, 0) = " bien";      # "crèm biene brûlée" 或 "crème bien brûlée"  
$cb =~ s/^\X{5}\K/ bien/;      # "crème bien brûlée"
```

可以注意到基于码点的方法是不可靠的。字符串采用NFD时会得到第一个答案，采用NFC时会得到第二个答案。你可能认为保持为NFC或者把它置为NFC就能解决你的所有问题，但实际上并非如此。一方面，没有预作形式的字形要远远多于有预作形式的字形，所以NFC根本无法保证不出现组合标记。

另外，NFC实际上更难处理，这也是为什么我们建议将输入规范化为NFD。考虑一下如何找出包含两个e的单词，如“crème”和“brûlée”。最简单也是唯一可靠的方法是：

```
/ e .*? e /x
```

这只适用于NFD，而不适用于NFC。你可能认为如果能保证NFC，那么可以写为：

```
/ [éeè] .*? [éeè] /x
```

不过对于crêpes这就不行了。增加一个ên只是看起来有帮助，因为很快你就会得到类似下面这样可怕的模式：

```
/ [èéëëëëëëëëëë] [èéëëëëëëëëëë] /x # 连续两个e
```

如果有一个加下划线的e，这个模式将无法工作，因为没有相应的预作字符。如果（当且仅当）你的字符串采用NFD，那么下面这个模式是可以的：

```
/ (?:(?=e) \X ){2} /x
```

这个模式提供了一种可靠而且非破坏的方式来完成不区分重音符的匹配：用\X匹配字形，并做出限制，要求必须以你查找的字形基本字符开头。如果采用这种方法，即首先通过NFD（或者可能NFKD）规范化所有内容，那么那些不分解的字母将无法得到，因为它们本身就是字母。

例如，你无法在“smørrebrød”中找到o，因为LATIN SMALL LETTER O WITH STROKE没有分解形式可以分离出o。尽管在“Ævar Arnfjörð Bjarmason”的分解形式中可以找到o，不过

找不到任何e或d，因为LATIN CAPITAL LETTER AE不会分解为一个a和一个e，而LATIN SMALL LETTER ETH不会转换为一个d。

至少使用规范化分解时无法找到这种不分解的字母。不过，使用Unicode::Collate集合的collator对象进行比较时，尽管只想检查主要成分，但实际上这3个字母都会找到。在下一节“Unicode文本比较和排序”中，我们将介绍这是如何做到的。

如果每次都要按照\X改造Perl的内置字符串函数，这会有些麻烦。另一种候选方法是使用Unicode::GCString CPAN模块。常规的Perl字符串都是面向码点的，不过Unicode::GCString模块是面向对象的，它允许将Unicode字形簇字符串作为字形来访问。下面展示了如何使用这个模块的方法像前面那样处理一个字形串：

```
my $gs = Unicode::GCString("crème brûlée");

say $gs->length();
say $gs->substr(0,5);
$gs->substr(-6, 6, "fraîche");
$gs->substr( 5, 0, " bien");
```

这里范式并没有影响，所以length方法会按字形返回答案，substr方法会按字形处理，你甚至可以使用index和rindex方法搜索直接量子串，这会按字形而不是按码点得到整数偏移量。

在这个模块中，最有用的方法可能是columns。假设你想打印一些菜单项，如下：

crème brûlée	£5.00
trifle	£4.00
toffee ice cream	£4.00

如何对齐呢？再假设你在使用一种定宽字体，不能使用下面的代码：

```
printf("%-25s £%.2f\n", $item, $price);
```

因为Perl认为每个码点都只有一列，但实际上并非如此。

columns方法会告诉你打印出一个字符串实际上会占多少个打印列。通常这与按字形统计的字符串长度相同，不过也有可能不同。Unicode认为一些字符是“宽字符”，它们在打印时要占两列。这种字符在东亚文字系统中非常常见，所以Unicode提供了类似East\_Asian\_Width=Wide和East\_Asian\_Width=Full的属性，指示这些字符要占两个打印列。

还有一些字符根本不占打印列，并不只是因为它们是组合标记：也有可能是控制或格式字符。另外，有些组合标记被认为是间隔标记，这实际上会占据打印列。唯一可以确信的是：如果采用一种定宽字体，每个字符的宽度肯定是列宽的整数倍。

要想打印一个填充为某个指定宽度的字符串，以下代码给出了一种方法：



```

sub pad {
    my($s, $width) = @_ ;
    my $gs = Unicode::GCString->new($s);
    return $gs . ( " " x ($width - $gs->columns));
}

printf( "%s £%.2f\n", pad($item, 25), $price);

```

现在即使你的字符串中包含格式化字符、组合标记或宽字符，列也能对齐。

可以看到，Unicode::GCString很有趣，也很有用，不过实际上它只是一个辅助模块，用来帮助一个更大的模块解决更困难的问题，这个更大的模块就是CPAN的Unicode::LineBreak模块。这个模块实现了Unicode换行算法（UAX#14）。如果你想把你的Unicode文本格式化为类似Text::Autoformat模块Unix *fmt*(1)程序的段落，就必须使用这个算法。CPAN Unicode::Tussle的*unifmt*程序就是这样一个例子。即使面对东亚宽字符、制表符、组合字符和不可见的格式码，它也能正常工作。

## Unicode文本比较和排序

使用Perl的内置sort或cmp操作符时，并不是按字母比较字符串。实际上，会把一个字符串中各个字符的数值码点与另一个字符串中相应字符的数值码点进行比较。如果文本中有些字母是多个语言共有的，而有些字母是各个语言所特有的，那么采用这种方式比较文本就不合适了。这不只是因为字母的码点混乱（毕竟，数字和其他可能连续的序列也存在这个问题），而是因为有些字母在字符集很小时就已经加入，而另外一些字母则是在字符集发展壮大之后才增加，如Topsy。例如，平方和立方很早就增加到Latin-1中。注意它们在排序时也排在最前面：

```

use v5.14;
use utf8;
my @exes = qw( x7 x0 x8 x3 x6 x5 x4 x2 x9 x1 );
@exes = sort @exes;
say "@exes";
# 打印: x2 x3 x1 x0 x4 x5 x6 x7 x8 x9

```

由于码点顺序并不对应字母表顺序，你的数据最后就会呈现这种顺序，尽管这不算是随机顺序，不过肯定不是我们所希望的字典顺序。这种默认排序顺序的好处主要体现在可以快速得到一个不变的顺序，尽管这不是字母表顺序。但它确实是确定的。有些情况下这已经足够了，但有些情况下可能还不行……

这就引入了标准Unicode::Collate模块，它实现了Unicode排序算法（UCA），这是特别为Unicode数据设计的一个高度定制的多层排序算法。这个模块有很多有意思的特性，不过通常只会调用它的默认sort方法：

```

use v5.14;

```

```

use utf8;
use Unicode::Collate;
my @exes = qw( x7 x0 x8 x3 x6 x5 x4 x2 x9 x1 );
@exes = Unicode::Collate->new->sort(@exes);
say "@exes";
# 打印: x0 x1 x2 x3 x4 x5 x6 x7 x8 x9

```

默认地，这个模块会提供一个字母数字顺序。大致说来，这就像先扔掉所有非字母数字字符，然后对余下的字符完成不区分大小写的排序，并不是根据数值码点顺序，而是按照字符串中字母表的顺序。这正是字典使用的排序，所以有时把它称为字典排序。

计算机并不理解如何对文本排序，不过在人们习惯于这一点之前，这就是人们所期望的排序方式，而且现在仍然是这样。如果一个书名中第一个单词后面有一个逗号，而另一个书名没有逗号但其他单词都相同，这两个书名应该排在一起，而不会分别排在不同的位置。逗号不应有影响，至少在尝试比较过所有其他字符之后才会考虑逗号。任何自然序列（如字母或数字）都不包括逗号。

考虑使用Perl的内置sort会发生什么（这与shell命令行和大多数编程语言中的默认字符串排序操作相同）：

```

% perl -e 'print sort <>' little-reds
Little Red Mushrooms
Little Red Riding Hood
Little Red Tent
Little Red, More Blue
Little, Red Rider

```

为什么会这么乱七八糟？“More”本应该在“Mushrooms”前面，“Rider”和“Riding”应该在一起，“Tent”本应该在最后面。即使采用ASCII，也不是按字母顺序排序。不过下面可以按字母顺序排序：

```

% perl -MUnicode::Collate -e '
    print for Unicode::Collate->new->sort(<>)' little-reds
Little Red, More Blue
Little Red Mushrooms
Little, Red Rider
Little Red Riding Hood
Little Red Tent

```

在我们看来，你肯定很喜欢Unicode的sort，所以可能想保留一个小的文字系统对常规文本进行排序。下面假设提供UTF-8输入，并生成UTF-8输出：

```

#!/usr/bin/perl
use warnings;
use open qw(:std :utf8);
use warnings qw(FATAL utf8);
use Unicode::Collate;
print for Unicode::Collate->new->sort(<>);

```

ucsort程序（这是CPAN Unicode::Tussle的一部分）中还提供了这个程序的另一个有更多特性的版本。

很多人都发现，如果采用其默认方式，这个模块的sort能生成很美观的结果。它很清楚如何对字母和数字排序，还了解扰乱ASCII排序的所有Unicode怪异字符（如数值不接近但排序时要排在一起的字母）、所有奇怪的Unicode大小写规则、按规范等价的字符串，以及很多其他方面。

另外，如果这不是你喜欢的排序方式，也可以自行定制，而且定制的潜力是无止境的。下面是一个简单的调整，对于英语的书名或影片名很适用。这一次我们把大写排在小写前面，而且排序之前先删除评论文章，对左边的数字完成“0填充”（zero-pad），所以101 Dalmations会排在7 Brides for 7 Brothers后面。<sup>注7</sup>

```
my $collator = Unicode::Collate->new(
    -upper_before_lower => 1,
    -preprocess => {
        local $_ = shift;
        s/^(?: The | An? ) \h+ //x; # 删除文章
        s/ ( \d+ ) / sprintf "%020d", $1 /g;
        return $_;
    };
);
```

我们已经知道，对于ASCII而言，按字母表顺序排序看起来比码点排序要好。对于非ASCII的Unicode，反差甚至更大。尽管你“只”使用英语，不过仍然需要处理ASCII以外的其他字符。如果你的数据中有一个10 ¢的邮票或者一个£ 5的支票该怎么办？即使在纯英语文本中，也会映射大括号、奇怪的短横线以及ASCII不能处理的所有专用符号。即使我们只讨论可以在英语字典里找到的单词，但也不能避免这种问题。以下只是从牛津英语字典里摘取的部分非ASCII辞条，这里以默认模式使用UCA排序（以列为主）：

Allerød	fête	Niçoise	smørrebrød
après-ski	feuilleté	piñon	soirée
Bokmål	flügelhorn	plaçage	tapénade
brassière	Gödelian	prêt-à-porter	vicuña
caña	jalapeño	Provençal	vis-à-vis
crème	Madrileño	quinceañera	Zuñi
crêpe	Möbius	Ragnarök	α-ketoisovaleric acid
désœuvrement	Mohorovičić discontinuity	résumé	(α-)lipoic acid
Fabergé	moiré	Schrödinger	(β-)nornicotine
façade	naïve	Shijō	ψ-ionone

---

注7： 这个填充是必要的，因为尽管Unicode知道单个数值码点的数值，但它不知道“9”应当在“10”前面（除非你像这样填充）。



如果这些单词与只采用ASCII的类似单词一同排序，你肯定不想看到它的排序结果（这个结果可不怎么样）。而这只是拉丁文本而已。考虑下面这些希腊神话里的人物，使用默认的码点顺序排序时可以得到：

Δύσις	Άσβολος	Διόνυσος	Φάντασος	Μεγαλήσιος
Ασβετος	Αγχίσης	Έσπερίς	Άγδιστις	Τελεσφόρος
Ασωπός	Λάχεσις	Έσπερος	Άστραίος	Χρυσόθεμις
Θράσος	Νέμεσις	Εύνοστος	Ασκληπιός	Άριστόδημος
Ιάσιος	Περσεύς	Ήφαιστος	Ήφαιστος	Άριστόμαχος
Νέσσος	Άδραστος	Ηωσφόρος	Άρισταίος	Λαιστρυγόνες
Πέρσης	Άλκηστις	Θρασκίας	Άσκάλαφος	
Πίστις	Αίγισθος	Πάσσαλος	Βορυσθενίς	
Χρύσος	Αργέστης	Πρόφασις	Έσπερίδες	

即使你不懂希腊字母表，也能看出按码点排序情况会有多糟糕：只需要从上向下查看每一列中的第一个字母。看到了吗？这些字母总在跳来跳去。如果采用默认的UCA排序，现在才能得到正确的排序结果：

Άγδιστις	Ασβετος	Έσπερίδες	Ιάσιος	Πίστις
Αγχίσης	Άσβολος	Έσπερίς	Λαιστρυγόνες	Πρόφασις
Άδραστος	Άσκάλαφος	Έσπερος	Λάχεσις	Τελεσφόρος
Αίγισθος	Ασκληπιός	Εύνοστος	Μεγαλήσιος	Φάντασος
Άλκηστις	Άστραίος	Ήφαιστος	Νέμεσις	Χρυσόθεμις
Αργέστης	Ασωπός	Ήφαιστος	Νέσσος	Χρύσος
Άρισταίος	Βορυσθενίς	Ηωσφόρος	Πάσσαλος	
Άριστόδημος	Διόνυσος	Θρασκίας	Περσεύς	
Άριστόμαχος	Δύσις	Θράσος	Πέρσης	

相信了吧？下面先来看UCA到底是如何工作的，然后考虑如何配置。

Unicode排序算法（UCA）是一种多层排序算法。你在前面已经看到。就好像你在写一个比较函数，将传入内置sort操作（也就是说，排序时将使用这个比较函数进行比较），如下所示：

```
@collated_text = sort {
    primary($a)    <=> primary($b)
    ||
    secondary($a)  <=> secondary($b)
    ||
}
```

```

    tertiary($a) <=> tertiary($b)
    ||
    quaternary($a) <=> quaternary($b)

} @random_text;

```

这是一个多层排序，从某种简化的角度来讲，这正是UCA所做的。这4个函数分别返回一个数，表示对这个成分的排序权值。只有当主要成分相同时才会继续比较次要成分，如此继续。

这有点简化，不过基本上就是这样做的。

### 主要成分：比较字母

比较基本字母<sup>注8</sup>是否相同。这个阶段要忽略非字母，可以跳过所有非字母，直到找到字母。如果相同相对位置上的字母不同，可以根据已有的字典顺序确定哪一个字母在前。

如果你使用拉丁字母表对拉丁文本排序，这就是你在学校里学到的abc顺序，所以“Fred”在“freedom”前面，“free beer”也在“freedom”前面。“free beer”之所以在“freedom”前面，这是因为第一个字符串中的第五个字母是“b”，而第二个字符串中的第五个字母是“d”，“b”比“d”靠前。看到是怎么做的了吧？这就是字典顺序。这里不是按字段排序。

### 次要成分：比较变音符

如果字母相同，再检查变音符是否相同（理论上讲，变音符就是组合标记。变音符和标记大多数情况下是重叠的，不过也不完全重叠）。默认地，我们会从左向右查看变音符来解决这个问题，不过也可能翻转为从右向左，比如法语就要求这样（对此有一个经典的演示示例，正常的LTR（从左向右）顺序会得出cote < coté < côte < côté，而采用法语的RTL（从右向左）顺序会得到cote < côte < coté < côté。换句话说，中间两个单词在法语顺序中会交换位置。这与其屈折构词法有关，因为法语是以词尾为基础的）。

### 第三成分：比较大小写

如果字母和变音符都相同，再检查大小写是否相同。默认地，小写字母在大写字母前，不过构造你的collator（排序器）对象时，使用upper\_before\_lower => 1选项就可以很容易地翻转这个顺序。

### 第四成分：比较所有其他部分

如果给定位置的字母、变音符和大小写都相同，现在再回来重新考虑所有非字母，如标点符号和空白符，这些部分在之前的阶段中被暂时忽略了。而在这个阶段，所有部分都需要考虑。

---

注8： 以及数字，以及你可能没有意识到是字母的那些字符。说到字母时要认为我们说的是所有这些字符。

如果你不想完成所有4个阶段的比较，也是完全可以的。例如，你可以指出只使用主要成分，这就只考虑基本字母，而不会考虑其他部分。

使用collator对象的eq方法对字符串完成“不区分重音符”的比较时就是这样做的。

规范化并不一定总能提供足够的帮助。例如，不能利用规范化查看“o”，“ö”和“ø”是否相同，因为LATIN SMALL LETTER O WITH STROKE没有包含“o”的分解形式。另一方面，比较字母是否相同时，正常情况下Unicode::Collate确实会把“o”，“ö”和“ø”看作是相同的字母，不过瑞典语或匈牙利语中不是这样。

类似地，对于“d”和“ð”，不能把LATIN SMALL LETTER ETH分解为包含“d”的形式，不过UCA会把它们都当作相同的字母。嗯，除了冰岛语以外，在冰岛语中“d”和“ð”本身就是不同的字母。

如果你希望collator对象忽略大小写，但是要考虑第一层的重音符，可以设置为只完成前两个阶段，而跳过其余阶段，为此要为构造函数传入一个选项对level => 2。

下面是v0.81版本中构造函数的所有可选配置参数的完整语法：

```
$Collator = Unicode::Collate->new(
    UCA_Version => $UCA_Version,
    alternate => $alternate, # 'variable'的别名
    backwards => $levelNumber, # 或\@levelNumbers
    entry => $element,
    hangul_terminator => $term_primary_weight,
    ignoreName => qr/$ignoreName/,
    ignoreChar => qr/$ignoreChar/,
    ignore_level2 => $bool,
    katakana_before_hiragana => $bool,
    level => $collationLevel,
    normalization => $normalization_form,
    overrideCJK => \&overrideCJK,
    overrideHangul => \&overrideHangul,
    preprocess => \&preprocess,
    rearrange => \@charList,
    rewrite => \&rewrite,
    suppress => \@charList,
    table => $filename,
    undefName => qr/$undefName/,
    undefChar => qr/$undefChar/,
    upper_before_lower => $bool,
    variable => $variable,
);
```

可以参考这个模块的手册页来了解构造函数接受的各个参数。尽管这个模块内置于Perl，不过也可以作为CPAN模块。所以，它能独立于Perl内核进行更新。随Perl v5.14一起发布的是v0.73版本的Unicode::Collate，所以很显然，在那之后它已经有更新。要运行最新版



本的Unicode::Collate模块，并不要求你有最新版本的Perl。它甚至支持v5.6的Perl，通过其UCA\_Version构造函数参数可以提供对之后Unicode版本的向前兼容性。

## 结合使用UCA 和Perl的sort

实际代码中，往往采用两种方式调用sort内置函数。第一种方式是调用时不提供任何排序例程，第二种方式则提供一个块参数作为定制比较函数。Unicode::Collate的sort方法可以用第一种方式来调用，但不能用第二种方式调用。要想指定定制的比较函数，需要使用collator对象的另一个方法，名为getSortKey。

假设一个程序中使用了内置sort函数，如下：

```
@srecs = sort {
    $b->{AGE} <=> $a->{AGE}
    ||
    $a->{NAME} cmp $b->{NAME}
} @recs;
```

不过你希望NAME字段按字母顺序排序，而不是按数值码点排序。为此，可以要求collator对象返回最终需要排序的各个文本串的二进制排序键。不同于规则文本，如果将这个二进制排序键传递给cmp操作符，它就会像有魔法一样按你希望的顺序排序。

传入sort的代码块如下所示：

```
my $collator = Unicode::Collate->new();
for my $rec (@recs) {
    $rec->{NAME_key} = $collator->getSortKey( $rec->{NAME} );
}
@srecs = sort {
    $b->{AGE} <=> $a->{AGE}
    ||
    $a->{NAME_key} cmp $b->{NAME_key}
} @recs;
```

可以向构造函数传入可选参数来完成一些特殊的处理，包括预处理。

还可以使用collator对象完成不区分重音符和大小写的简单匹配。这是有道理的，既然能区分什么情况下是有序的，那么也应该能指出根据给定的顺序在什么情况下是等价的。所以只需要选择正确的顺序语义。例如，如果将排序层次设置为1，它只考虑是否是相同的字母，而不考虑重音符和大小写。对此，collator对象有一些方法（如eq、substr和index）可以提供帮助（不过，需要将它设置为不完成规范化，否则码点偏移量将是错的）。

下面给出一个例子：

```
use v5.14;
use utf8;
use Unicode::Collate;
```

```

my $Collator = Unicode::Collate->new(
    level => 1,
    normalization => undef,
);
my $full = "Gabriel García Márquez";
for my $sub (qw[MAR CIA]) {
    if (my($pos,$len) = $Collator->index($full, $sub)) {
        my $match = substr($full, $pos, $len);
        say "Found match of literal <$sub> in <$full> as <$match>";
    }
}

```

运行时会打印以下结果：

```

Found match of literal <MAR> in <Gabriel García Márquez> as <Már>
Found match of literal <CIA> in <Gabriel García Márquez> as <cía>

```

不要告诉CIA。

## 本地化排序

尽管默认的UCA方式对于英语和很多其他语言很适用（包括爱尔兰盖尔语、印尼语、意大利语、格鲁吉亚语、荷兰语、匈牙利语和德语，但德语的电话号码簿除外），不过对于讲其他语言的人来说，要按他们期望的字母表顺序排序还需要做一些修改。或者有时甚至没有字母表。

例如，北欧语言把一些有重音符的字母放在z后面，而不是放在相应的常规字母旁边。甚至西班牙语还有不同的做法：匈牙利语中认为 $\tilde{a}$ 和 $\tilde{o}$ 是常规字母上加一个波浪线，与之不同，西班牙语则认为 $\tilde{n}$ 并不是一个常规的 $n$ 上面加一个波浪线。相反，它有自己的字母（当然名为 $e\tilde{n}e$ ），这个字母在西班牙字母表中排在 $n$ 的后面并且在 $o$ 的前面。这说明在西班牙语中下面的单词会按以下顺序排序：*radio*, *ráfaga*, *ranúnculo*, *raña*, *rápido*, *rastrillo*。要注意*ranúnculo*在*raña*前面而不是后面。

如果要对Unicode数据完成特定于本地化环境的排序，需要使用Unicode::Collate::Locale模块。这是Unicode::Collate发行版本中的一部分，已经作为标准随v5.14提供，另外如果你从CPAN单独安装，CPAN也提供了相应的模块。

这两个模块的API之间唯一的区别是：Unicode::Collate::Locale的构造函数有一个额外的参数：本地化环境（locale）。写这本书时，已经支持70个不同的本地化环境，还包括一些变形，如德语的电话号码簿（变音元音排序时，就好像这些变音元音是常规元音后面加一个 $e$ ）、传统西班牙语（ $ch$ 和 $ll$ 作为字形在字母表中有自己的顺序）、日语以及5种不同的中文排序方法。

使用这些本地化环境很容易：

```
use Unicode::Collate::Locale;
$coll = Unicode::Collate::Locale->new(locale => "fr");
@french_text = $coll->sort(@french_text);
```

由于Unicode::Collate::Locale是Unicode::Collate的一个子类，它的构造函数也像其超类一样接受同样的可选参数，而且Locale对象支持同样的方法，所以你可以像从前一样使用这些对象完成本地化环境敏感地搜索。我们选择“德语电话号码簿”本地化环境，在这里（例如）*ae*和*ä*被看作是相同的字母。可以直接比较，如下所示：

```
state $coll = new Unicode::Collate::Locale::
    locale => "de__phonebook",
    ;

if ($coll->eq($a, $b)) { ... }
```

下面给出一种搜索方法：

```
use Unicode::Collate::Locale;
my $Collator = new Unicode::Collate::Locale::
    locale => "de__phonebook",
    level => 1,
    normalization => undef,
    ;

my $full = "Ich müß Perl studieren.";
my $sub = "MUESS";
if (my ($pos,$len) = $Collator->index($full, $sub)) {
    my $match = substr($full, $pos, $len);
    say "Found match of literal <$sub> in <$full> as <$match>";
}
```

运行时，会得到以下结果：

```
Found match of literal <MUESS> in <Ich müß Perl studieren.> as <müß>
```

## 更多内容

还有一点要注意，默认情况下，Perl快捷方式（如\w，\s甚至\d）会根据特定的字符属性匹配多个Unicode字符。如表5-6所示，它将匹配Annex C: Compatibility Properties（兼容性属性）给出的规范定义，这个定义见于Unicode技术标准#18“Unicode正则表达式”第13版（2008年8月发布）。

如果习惯通过匹配(\d+)来获取一个整数，并作为数字使用，这对于Unicode数据可不一定正确。在Unicode v6.0中，\d能匹配420个码点。如果这不是你希望的，可以指定/\d/a或/(?a:\d)/，或者可以使用更特定的属性\p{POSIX\_Digit}。

不过，如果你打算匹配任何文字系统中的任何数字，而且需要使用这个匹配结果作为Perl中的一个数，Unicode::UCD模块的num函数可以帮你做到。



```

use v5.14;
use utf8;
use Unicode::UCD qw(num);
my $num;
if ("४५६७" =~ /\d+/) {
    $num = num($1);
    printf "Your number is %d\n", $num;
    # 你的数字是4567
}

```

尽管正则表达式可以询问一个字符是否有某个属性，但是无法指出字符拥有哪些属性（至少需要测试所有属性后才可能告诉你）。不过有时你确实想知道字符有哪些属性。例如，假设你想知道一个码点指定为哪个文字系统，或者它的通用类别是什么。为此，还可以使用Unicode::UCD模块。下面的程序会打印模式匹配中可以使用的属性：

```

use v5.14;
use utf8;
use warnings;

use Unicode::UCD qw( charinfo );
use Unicode::Normalize qw( NFD );

## 取消下面这行代码的注释，得到分解的形式
my $mystery = ## NFD
               "७¾ζΩ";
for my $chr (split //, $mystery) {
    my $ci = charinfo(ord $chr);
    print "U+", $$ci{code};
    printf ' \N{%s}'. "\n\t", $$ci{name};
    print " gc=",          $$ci{category};
    print " script=",      $$ci{script};
    print " BC=",          $$ci{bidi};
    print " mirrored=",    $$ci{bidi};
    print " ccc=",         $$ci{combining};
    print " nv=",          $$ci{numeric};
    print "\n";
}

```

运行时，这个程序会打印以下结果：

```

U+096D \N{DEVANAGARI DIGIT SEVEN} gc=Nd script=Devanagari
      BC=L mirrored=L ccc=0 nv=7
U+00BE \N{VULGAR FRACTION THREE QUARTERS} gc=No script=Common A
      BC=ON mirrored=ON ccc=0 nv=3/4
U+00E7 \N{LATIN SMALL LETTER C WITH CEDILLA} gc=Ll script=Latin
      BC=L mirrored=L ccc=0 nv=
U+1F6F \N{GREEK CAPITAL LETTER OMEGA WITH DASIA AND PERISPOMENI}
      gc=Lu script=Greek BC=L mirrored=L ccc=0 nv=

```

不过，如果删除注释阻止NFD运行，会得到：

```

U+096D \N{DEVANAGARI DIGIT SEVEN} gc=Nd script=Devanagari
      BC=L mirrored=L ccc=0 nv=7

```

```

U+00BE \N{VULGAR FRACTION THREE QUARTERS} gc=No script=Common
      BC=ON mirrored=ON ccc=0 nv=3/4
U+0063 \N{LATIN SMALL LETTER C} gc=Ll script=Latin
      BC=L mirrored=L ccc=0 nv=
U+0327 \N{COMBINING CEDILLA} gc=Mn script=Inherited
      BC=NSM mirrored=NSM ccc=202 nv=
U+03A9 \N{GREEK CAPITAL LETTER OMEGA} gc=Lu script=Greek
      BC=L mirrored=L ccc=0 nv=
U+0314 \N{COMBINING REVERSED COMMA ABOVE} gc=Mn script=Inherited
      BC=NSM mirrored=NSM ccc=230 nv=
U+0342 \N{COMBINING GREEK PERISPOMENI} gc=Mn script=Inherited
      BC=NSM mirrored=NSM ccc=230 nv=

```

## 定制正则表达式边界

\b表示单词边界，\B表示非单词边界，不过这些都依赖于\w当前的定义（这说明，如果你用/a和/aa修饰符切换为ASCII语义，它们就会立刻改变）。

如果这不是你想找的边界，还可以根据任意边界条件编写你自己的边界断言，如文字系统边界。下面是\b的定义：

```

(?(?<= \w) # 如果左边有一个单词字符
  (?! \w)   # 那么右边不能有单词字符
  | (?= \w)  # 否则右边必然有一个单词字符
)

```

以下是\B的定义：

```

(?(?<= \w) # 如果左边有一个单词字符
  (?= \w)   # 那么右边必然有一个单词字符
  | (?! \w)  # 否则右边不能有单词字符
)

```

既然已经知道单词边界和非单词边界是如何工作的，在以上模式中看到\w的地方可以换入你自己的条件，来建立你自己的边界。只是要注意，需要指定一个定宽的条件，从而能够在前瞻断言中使用。这说明不能使用类似\X或\R的序列，因为这些序列的宽度是可变的。最容易的做法是使用一个属性或其他字符类。例如，可以使用\p{Greek}表示希腊文字中的字符。不过最好增加Inherited，这样就不会漏掉组合字符，所以应当使用[\p{Greek}\p{Inherited}]。

例如，可以如下提供正则表达式子例程来完成这种工作：

```

(? (DEFINE)
  (?<greeklish> [\p{Greek}\p{Inherited}])
  (?<ungreeklish> [^\p{Greek}\p{Inherited}])
  (?<greek_boundary>
    (?(?<= (?&greeklish))
      (?! (?&greeklish))
      | (?= (?&greeklish))
    )
  )
)

```

```

    )
  )
  (?<greek_nonboundary>
    (?(?<=      (?&greeklish))
      (?=      (?&greeklish))
      | (?!      (?&greeklish))
    )
  )
)

```

有些字符类是由于增加和减少现有Unicode属性得到的，或者是将Unicode属性取反以及相交得到的，如上面的<greeklish>正则表达式子例程，你可能希望把这些字符类实现为定制属性。定制属性看起来与普通属性很类似。例如：

```

sub IsGreeklish {
    return <<'END';
+utf8::IsGreek
+utf8::IsInherited
END
}

```

只要模式在该子例程所在的包中编译，现在就可以使用\p{IsGreeklish}和\P{IsGreeklish}。具体如何使用请参见下一节。

## 构建字符

要定义你自己的属性，首先需要写一个子例程，要与你想要的属性同名（参见第7章）。出于安全方面的原因，子例程的（非限定）名必须以Is或In开头。这个子例程应当在需要该属性的包中定义（参见第10章），这说明，如果希望在多个包中使用这个属性，就必须从一个模块中导入这个子例程（见第11章），或者将它作为一个类方法从定义这个子例程的包继承（见第12章）。

一旦解决上述问题，这个子例程就要返回数据，其格式与PATH\_TO\_PERLLIB/unicode/Is目录中的文件格式相同。也就是说，以十六进制数形式返回一组字符或字符范围，每行一个字符（或字符范围）。如果有范围，那么作为范围上下界的两个数字之间用一个制表符分开。假设你希望有这样一个属性：如果字符在任何一个日文假名表中，即平假名或片假名（统称为假名），则这个属性为true。可以如下加入这两个范围：

```

sub InKana {
    return <<'END';
3040    309F
30A0    30FF
END
}

```

或者，也可以按现有的属性名来定义：

```

sub InKana {

```



```

        return <<'END';
+utf8::InHiragana
+utf8::InKatakana
END
}

```

还可以使用一个“-”前缀设置“减法”。假设你只希望得到具体的字符，而不包括字符范围。可以如下去除所有未定义的字符：

```

sub IsKana {
    return <<'END';
+utf8::InHiragana
+utf8::InKatakana
-utf8::IsCn
END
}

```

还可以使用“!”前缀从一个求补字符集开始：

```

sub IsNotKana {
    return <<'END';
!utf8::InHiragana
-utf8::InKatakana
+utf8::IsCn
END
}

```

相交用“&”前缀指定，这对于获取两个（或多个）类匹配的共同字符很有用。

```

sub IsGraecoRomanTitle {<<'END_OF_SET'}
+utf8::IsLatin
+utf8::IsGreek
&utf8::IsTitle
END_OF_SET

sub IsGreekTitle {<<'END_OF_SET'}
+main::IsGraecoRomanTitle
-utf8::IsLatin
END_OF_SET

```

要记住重要的一点，不要对第一个字符集使用“&”；这会与空相交，最终会得到一个空集。

在你自己的定制字符类中（如`[-.\w\s]`）包含Perl的“经典”字符类（如`\w`）时，Perl也使用同样的技巧定义这些字符类的含义。你可能认为你的规则越复杂，它们运行速度就越慢。不过，实际上，Perl已经为属性的一个特定的64位swatch块计算了位模式，并将其缓存，从而不用再次重新计算这个模式（它是在64位swatch块中计算的，所以甚至不用解码UTF-8就可以完成查找）。因此，所有字符类（不论内置字符类还是定制字符类），一旦开始运行，速度基本上就是一样（快）的。

如果希望改变中括号字符类的语法来完成定制，有关的内容可以参考CPAN `Unicode::Regex::Set`模块。

结合定制名，利用定制属性甚至可以管理私用的码点，而不用求助于丑陋的数字。例如，Unicode还没有把谈格瓦文字（Tengwar，一种精灵文字）加入其官方字符库中（不过，已经列入路线图，毕竟中土世界有很多文字映射）。字体设计者们并没有因此停下脚步，他们还会继续创建美丽而有用的谈格瓦文字字体。尽管有些字体确实使用了Unicode为谈格瓦文字保留的码点块，不过大多数都会使用一个私用范围的码点。但不管怎样，这些码点都没有指定的名字或属性。

这不会成为Perl的障碍，因为很容易为字符和属性创建你自己的名字。Perl中已有的Tengwar模块提供了如下命名字符：

TENGWAR LETTER TINCO	TENGWAR DIGIT ZERO
TENGWAR LETTER PARMA	TENGWAR DIGIT ONE
TENGWAR LETTER CALMA	TENGWAR DIGIT TWO
TENGWAR LETTER QUESSE	TENGWAR DIGIT THREE

基于这些命名字符，可以很顺利地编写类似下面的代码：

```
if ($elvish =~ /\N{TENGWAR LETTER SILME NUQUERNA}/) {...}
```

甚至可以对一个Tengwar码点使用`charnames::viacode`来得到它的定制名。更棒的是，它还提供了如下Tengwar字符属性：

In_Tengwar	In_Tengwar_Numerals
In_Tengwar_Consonants	Is_Tengwar_Decimal
In_Tengwar_Vowels	Is_Tengwar_Duodecimal
In_Tengwar_Alphabetics	In_Tengwar_Marks
In_Tengwar_Punctuation	In_Tengwar_Alphanumerics

利用这些字符属性，可以写出以下Perl代码：

```
print "W" if /\p{In_Tengwar_Alphanumerics}/;
print "A" if /\p{In_Tengwar_Alphabetics}/;
print "C" if /\p{In_Tengwar_Consonants}/;
print "V" if /\p{In_Tengwar_Vowels}/;
```

或者甚至可以有以下代码：

```
$TENGWAR_GRAPHHEME = qr{
    (?>
        (?= \p{In_Tengwar} ) \P{In_Tengwar_Marks}
        \p{In_Tengwar_Marks} *
    ) | \p{In_Tengwar_Marks}
}x;
```

如果要写这种代码，而不能对抽象（包括字符和属性）命名，这有点像尝试为计算机写一个程序，其中只使用数值内存地址而没有变量名。当然，如果你非常想这样做，也是可以

的，不过这样就不能顺利地融入现有的功能，而且可能不可读或不可维护。Perl允许为这种定制应用（私用领域）创建自己的特殊用途语言，从而可以帮助你编写简单而清晰的代码。

## 参考资料

Perl很努力地密切跟踪着最新发布的Unicode标准。这些标准包括各个附件和技术报告。对于本章中讨论的资料，适用的标准包括：

*UAX #44: Unicode Character Database* (Unicode字符数据库)

*UTS #18: Unicode Regular Expressions* (Unicode正则表达式)

*UAX #15: Unicode Normalization Forms* (Unicode范式)

*UTS #10: Unicode Collation Algorithm* (Unicode排序算法)

*UAX #29: Unicode Text Segmentation* (Unicode文本切分)

*UAX #14: Unicode Line Breaking Algorithm* (Unicode换行算法)

*UAX #11: East Asian Width* (东亚字符宽度)



# 子例程

与很多语言类似，Perl也支持用户自定义的子例程<sup>注1</sup>。这些子例程可以在主程序的任何位置定义，也可以通过do、require或用use关键字从其他文件加载，或者可以使用eval在运行时生成。甚至可以在运行时使用第10章“自动加载”一节中介绍的机制加载子例程。可以使用一个包含子例程名或子例程引用的变量来间接调用子例程，或者通过一个对象来间接调用子例程，让对象确定应当调用哪一个子例程。还可以生成匿名子例程，匿名子例程只能通过引用来访问，如果你愿意，还可以通过闭包（将在第8章中的“闭包”一节介绍）使用这些匿名子例程来克隆几乎完全相同的新函数。

## 语法

要声明但不定义一个命名子例程，可以使用以下形式：

```
sub NAME
sub NAME PROTO
sub NAME      ATTRS
sub NAME PROTO ATTRS
```

要声明并定义一个命名子例程，需要增加一个代码块（BLOCK）：

```
sub NAME          BLOCK
sub NAME PROTO    BLOCK
sub NAME      ATTRS BLOCK
sub NAME PROTO ATTRS BLOCK
```

要创建一个匿名子例程或闭包，需要省略声明中的例程名（NAME）：

---

注1： 我们也称之为函数（functions），不过在Perl中函数与子例程是一样的。有时我们甚至把它们叫做方法（methods），子例程和文法用同样的方式定义，不过调用方式不同。

```

sub                                BLOCK
sub    PROTO                      BLOCK
sub                                ATTRS BLOCK
sub    PROTO ATTRS                BLOCK

```

*PROTO*和*ATTRS*表示原型和属性，将分别在本章后面的相应节中讨论。不过原型和属性并不重要，最重要的部分是*NAME*和*BLOCK*（尽管有时可以省略*NAME*和*BLOCK*）。

如果声明中没有指定*NAME*，那么必须提供一种调用子例程的方法。所以一定要保存返回值，因为这种形式的*sub*声明不仅会如你期望的那样在编译时编译，还会生成一个运行时返回值：

```
$subref = sub BLOCK;
```

要导入另一个模块中定义的子例程，可以采用以下方法：

```
use MODULE qw(NAME1 NAME2 NAME3 ...);
```

要直接调用子例程，可以使用以下方法：

```

NAME(LIST)      # 有小括号时&是可选的。
NAME LIST      # 如果子例程已经预声明或预导入，那么小括号是可选的。
&NAME          # 将当前@_提供给该子例程，
               # （并避开原型）。

```

要间接调用子例程（按名或按引用调用），可以采用以下任何一种形式：

```

&$subref(LIST)  # 对于间接调用，&是必不可少的。
$subref->(LIST)  # （除非使用中缀记法）。
&$subref        # 将当前@_提供给这个子例程。

```

子例程的正式名字包括&前缀。调用子例程时可以使用这个前缀，不过&通常是可选的，如果已经预先声明了这个子例程，那么小括号也是可选的。不过，如果为子例程命名，如将子例程用作为*defined*或*undef*的一个参数，或者希望用*\$subref = \&name*生成一个命名子例程的引用，这种情况下&就并非可选，而是不可缺少的。如果你想使用*&\$subref()*或*&{\$subref}()*构造建立一个间接子例程调用，&也是不可省略的。不过，如果采用更方便的*\$subref->()*记法，这种调用不要求有&。关于子例程引用的更多内容请参见第8章。

Perl对于子例程名是否使用大写并没有强制要求。不过，对此有一个不太严格的约定，Perl的运行时系统间接调用的函数（如*BEGIN*、*UNITCHECK*、*CHECK*、*INIT*、*END*、*AUTOLOAD*、*DESTROY*和第14章提到的所有函数）都使用大写，所以你的子例程名可能要避免使用大写（但是如果子例程用于提供常量值，这些子例程的名字通常也是大写。这应该没问题。希望如此）……

## 语义

先不要过于纠结上面的语法，只要记住，要定义一个简单的子例程，通常会这样做：



```
sub razzle {
    print "Ok, you've been razzled.\n";
}
```

而调用这样一个子例程，只需要：

```
razzle();
```

在这里，我们忽略了输入（参数）和输出（返回值）。不过，Perl中向子例程传入数据以及从子例程传出数据的模型相当简单：所有函数参数都作为一个扁平的标量列表传递，如果有多个返回值，也以类似的方式作为一个扁平的标量列表返回给调用者。与所有`LIST`一样，传入这些列表的数组或散列会把它们的值内插到扁平化的列表中，而失去它们原来的“身份”，不过有很多方法可以避免这一点，自动列表内插通常很有用。参数表和返回值列表可以根据你的需要包含任意多个标量元素（不过可以使用原型对参数表施加约束）。实际上，Perl就是基于这种变参函数的概念设计的（变参函数可以取任意多个参数），这与C不同，在C中只是以很勉强的方式拼凑实现了变参函数，从而能调用`printf(3)`。

现在如果你想围绕这个概念设计一种语言，允许传递数目不定的任意参数，最好能很容易地处理这些任意的参数表。Perl例程的参数都会作为数组`@_`传入。如果调用一个有两个参数的函数，在这个函数中，这两个参数可以作为数组`@_`的前两个元素来访问，即`$_[0]`和`$_[1]`。由于`@_`只是一个有非常规名字的常规数组，所以可以像任何其他数组一样正常处理`@_数组`<sup>注2</sup>。数组`@_`是一个局部数组，不过它的值是实际标量参数的别名（这称为“按引用传递”或“传引用”）。因此，如果修改了`@_`的相应元素，也会同时修改实际参数（不过，很少会这样做，因为在Perl中可以很容易地返回感兴趣的值）。

子例程（或任何其他代码块）的返回值是最后计算的表达式的值。或者，可以使用一个显式的`return`语句来返回最后的返回值，而且可以从子例程的任意位置退出子例程。不论采用哪一种方法，子例程在一个标量或列表上下文中调用时，例程中最后计算的表达式也会在同一个标量或列表上下文中计算。

## 关于参数表的技巧

Perl没有命名的形参，但在实际中你可能会把`@_`的值复制到一个`my`列表，这就相当于一个形参列表（复制值会把“按引用传递”的语义变成“按值传递”，这并不是偶然，人们通常就希望参数按值传递，即使他们并不知道这些计算机科学术语）。下面给出一个典型的例子：

```
sub maysetenv {
    my($key, $value) = @_;
    $ENV{$key} = $value unless $ENV{$key};
}
```

---

注2： 在这个方面，Perl比一般的编程语言更有正交性。



不过并不要求对参数命名，这正是@\_数组的关键。例如，要计算一个最大值，可以直接迭代处理@\_：

```
sub max {
    my $max = shift(@_);
    for my $item (@_) {
        $max = $item if $max < $item;
    }
    return $max;
}

$bestday = max($mon,$tue,$wed,$thu,$fri);
```

对于参数表很短的函数，这种位置参数是适用的，但随着参数个数的增加，要想记住哪个参数做什么，哪些可选，或者哪些有默认值，这会很困难。要解决所有这些问题，一种更灵活的方法是调用者使用“参数名/参数值”对来提供参数。每个名值对中的第一个元素是参数名；第二个是参数的值。这会使代码很明了，因为你不用读整个函数定义就能看出参数要用来做什么。更棒的是，其他程序员使用你的函数时，他们不必记住参数的顺序，而且可以不指定那些多余的、未使用的参数。我们强烈建议使用命名参数。

这里的技巧就是把@\_参数表赋至一个散列。

```
configuration(PASSWORD => "xyzzzy", VERBOSE => 9, SCORE => 0);

sub configuration {
    my %options = @_;
    print "Maximum verbosity.\n" if $options{VERBOSE} == 9;
}
```

为了显示这种做法的灵活性，下面给出一个例子，这是《Perl Cookbook》在“子例程”一章关于“按命名参数传递”一节中给出的例子。

```
thefunc(INCREMENT => "20s", START => "+5m", FINISH => "+30m");
thefunc(START => "+5m", FINISH => "+30m");
thefunc(FINISH => "+30m");
thefunc(START => "+5m", INCREMENT => "15s");
```

然后在这个子例程中创建一个散列，加载默认值以及命名对数组。

```
sub thefunc {
    my %args = (
        INCREMENT    => "10s",
        FINISH       => 0,
        START        => 0,
        @_,          # 实参默认默认值
    );
    if ($args{INCREMENT} =~ /m$/ ) { ... }
    ...
}
```

通过为每个参数值指定一个名字，然后把@\_赋至%args散列，你就不用再去努力记住参数的顺序，而且可以忽略一些参数（假定它们有某个默认值）。

另一方面，下面再给出一个例子，这里没有为形参命名，所以可以改变你的实际参数：

```
upcase_in($v1, $v2); # 这会改变$v1和$v2
sub upcase_in {
    for (@_) { $_ = uc($_) }
}
```

当然，不能以这种方式改变常量。如果参数实际上是一个标量直接量，如" hobbit"，或者是一个只读的标量变量，如\$1，倘若试图改变这个参数，Perl就会产生一个异常（这可能是致命的错误，有可能影响你的职业发展）。例如，下面这样做是不行的：

```
upcase_in("frederick");
```

如果把upcase\_in函数写为返回其参数的一个副本，而不是直接修改参数，这样可能会安全一些：

```
($v3, $v4) = upcase($v1, $v2);
sub upcase {
    my @parms = map { uc } @_;
    # 检查是否在列表上下文中调用这个函数
    return wantarray ? @parms : $parms[0];
}
```

注意，这个（没有原型）的函数并不关心传入的是真正的标量还是数组。Perl会把所有参数塞进一个很大很长的扁平@\_参数表中。这正是Perl的简单参数传递方式的一个闪光之处。无需改变upcase定义，它就能很好地工作，即使传入类似这样的参数，upcase函数也能正常处理：

```
@newlist = upcase(@list1, @list2);
@newlist = upcase( split /:/, $var );
```

不过，不要这么做：

```
(@a, @b) = upcase(@list1, @list2); # 不正确
```

为什么呢？因为与@\_中的扁平参数表类似，返回值列表也是扁平的。所以这会保存@a中的所有内容，并清空@b，即在@b中存储一个空列表。其他做法参见后面的“传引用”一节。

## 错误提示

如果希望函数以某种方式返回，使得调用者能够意识到存在一个错误，在Perl中要达到这个目的，最自然的方式就是使用一个裸return语句，即不带任何参数。这样一来，在标量上下文中使用这个函数时，调用者会得到undef；而在列表上下文中使用时，调用者将得到一个null（空）列表。

在特殊情况下，你还可以选择产生一个异常来指示错误。但是这种方法要少用；否则你的整个程序里就会充斥着大量异常处理器。例如，在一个通用的打开文件函数中，未能打开文件的情况绝不罕见。不过，这个错误有可能被忽略。如果函数在void上下文中调用，内置函数wantarray会返回undef，所以能判断出是否被忽略：

```
if ($something_went_away) {  
    return if defined wantarray; # 很好，不是void上下文  
    die "Pay attention to my error, you danglesocket!!!\n";  
}
```

## 作用域问题

子例程可以递归调用，因为即使是例程调用自身，每个调用也会得到它自己的实参数组。如果使用&形式调用一个子例程，参数表是可选的。倘若使用&而且忽略参数表，就会有特殊的情况发生：将隐式地提供调用例程的@\_数组。这是一个高效的调用机制，不过新用户最好避免这种用法。

```
&foo(1,2,3);      # 传入3个实参  
foo(1,2,3);       # 同样传入3个实参  
  
foo();            # 传入一个空列表  
&foo();          # 同样传入一个空列表  
  
&foo;             # foo()得到当前参数，类似于foo(@_)，不过速度更快！  
foo;              # 如果子例程foo已经预声明，则类似于foo()，否则就是裸字"foo"
```

&形式不仅可以省略参数表，还会禁用你为参数提供的原型检查。这是由于历史原因造成的，另一部分原因则是为了提供一个方便的“作弊”方法（如果你知道你在做什么）。有关内容参见这一章后面的“原型”一节。

对于函数中要访问的变量，如果没有将这些变量声明为该函数私有，并不要求它们一定是全局变量；这些变量仍遵循Perl的块作用域规则。在第2章的“名字”一节中已经解释过，这说明首先会在外围词法作用域中（或嵌套的多个外围词法作用域中）尝试解析，然后再查看包作用域。从子例程的角度来看，外围词法作用域的所有my或state变量仍是可见的。

例如，下面的bumpx函数可以访问文件作用域中的\$x词法作用域变量，因为声明my的作用域（即文件本身）在定义子例程之前没有结束：

```
# 文件开头  
my $x = 10;          # 声明和初始化变量  
sub bumpx { $x++ }   # 函数可以看到外围词法作用域变量
```

C和C++程序员可能会把\$x看作是一个“文件静态”变量。对于其他文件中的函数来说，它是私有的，但是在my之后声明的函数看来，这个变量是全局的。C程序员刚开始接触



Perl时，可能会查找文件或函数的“静态变量”，但是在Perl中找不到这个关键字。Perl程序员总在避免使用“静态”（static）这个词，因为静态系统没有生命，很乏味，另外也因为历史上这个词被大量滥用。

尽管Perl的字典中没有“static”这个词，但是Perl程序员完全可以利用一个类似的概念：`state`变量，来创建对函数私有而且可以跨函数调用保留的变量，这个内容将在后面详细解释。不过，这不是唯一的方法。Perl提供了更丰富的作用域原语，可以采用多种方式结合自动内存管理，那些寻找“static”关键字的人绝对想不到可以尝试这些用法。

词法作用域变量不会因为退出其作用域就自动被垃圾回收；而是要等待不再使用这些变量之后才会回收，这一点非常重要。要创建不会在函数调用之间自动重置的私有变量，需要把整个函数包围在一个额外的代码块中，并把`my`声明和函数定义都放在这个代码块中。甚至可以放入多个函数，从而能共同访问一个原本私有的变量：

```
{
    my $counter = 0;
    sub next_counter { return ++$counter }
    sub prev_counter { return --$counter }
}
```

与以往一样，对词法作用域变量的访问仅限于同一个词法作用域中的代码。另一方面，这两个函数的名字（在这个包中）可以全局访问，而且由于它们都在`$counter`的作用域中定义，所以尽管别处无法访问`$counter`变量，但这两个函数仍能访问。

如果这个函数是通过`require`或`use`加载的，可能没有问题。如果都在主程序中，则需要确保`my`的运行时赋值要足够早，为此，可以把整个代码块放在主程序前面，或者也可以把它包围在一个`BEGIN`或`INIT`块中，以确保它在程序启动之前先执行：

```
BEGIN {
    my @scale = ( "A" .. "G" );
    my $note = -1;
    sub next_pitch { return $scale[ ($note += 1) %= @scale ] };
}
```

`BEGIN`并不影响子例程定义，也不会影响子例程使用的所有词法作用域变量的持久存储。这个块的作用只是确保调用子例程之前先完成变量的初始化。关于声明私有和全局变量的更多内容，请参见第27章的“`my`”、“`state`”和“`our`”节。`BEGIN`和`INIT`构造将在第16章解释。

变量的声明应当贴近它的实际使用，要想更容易地做到这一点，“`state`”特性支持`my`关键字的一个变形。要启用这个特性，需要声明使用的Perl至少是v5.10版本。

现在可以使用`state`关键字来声明一个词法作用域变量只在第一次遇到时初始化：

```
use v5.14;
```

```
sub bumpx {
    state $x = 10;    # 只在第一次遇到时初始化
    return $x++;
}
```

现在这个函数的表现与前一个函数类似，先返回10，然后返回11，然后12，依此类推。下面这个函数中，有一个持久存储的私有散列来跟踪遇到一个变量的次数：

```
sub seen_count {
    state %count;
    my $item = shift();
    return ++$count{$item};
}
```

与其他变量声明不同，state变量的初始化仅限于简单的标量变量。数组和散列仍然可以作为state变量，不过不能像对标量那样对它们完成初始化。这看上去是一个限制，但实际上并没有限制，因为你完全可以存储你想要的那个类型的引用，这确实是标量。例如：

```
# 不能使用state %hash = (....)
my %hash = (
    READY => 1,
    WILLING => 1,
    ABLE => 1,
);
```

作为一个state变量，应当这样使用：

```
state $hashref = {
    READY => 1,
    WILLING => 1,
    ABLE => 1,
};
```

要使用state变量实现上面的next\_pitch函数，做法如下：

```
sub next_pitch {
    state $scale = ["A" .. "G"];
    state $note = -1;
    return $scale->[ ($note += 1) %= @$scale ];
}
```

state变量最关键的一点是：你不必使用一个BEGIN（或UNITCHECK）块来确保变量的初始化发生在函数调用之前。

最后一点，如果我们谈到state变量只初始化一次，这并不是说不同闭包中的state变量都是同一个变量。它们并不相同，所以每个state变量会有其自己的初始化。这正是state变量与其他语言中静态变量的不同之处。

例如，在下面两个版本的代码中，\$epoch是一个词法作用域变量，它对于返回的闭包是私

有的。不过，在`timer_then`中，它会在这个闭包返回之前初始化，而在`timer_now`中，会延迟到第一次调用返回的闭包时才完成`$epoch`的初始化：

```
sub timer_then {
    my $epoch = time();
    return sub {
        ...
    };
}

sub timer_now {
    return sub {
        state $epoch = time();
        ...
    };
}
```

## 传引用

如果希望将多个数组或散列传入或传出函数，而且希望它们保持其完整性，就需要使用一个显式的按引用传递（即传引用）机制。在此之前，你要先掌握引用（引用将在第8章详细介绍）。否则这一节对你意义不大。不过，嗯，可以先大致了解一下……

下面给出几个简单的例子。首先来定义一个函数，它需要一个数组的引用作为参数。数组很大时，作为一个引用传入函数比传入一个长长的值列表要快得多：

```
$total = sum ( \@a );

sub sum {
    my ($aref) = @_;
    my ($total) = 0;
    for (@$aref) { $total += $_ }
    return $total;
}
```

下面向函数传入多个数组，并使用`pop`将其分别弹出，最终返回一个新列表，其中包括之前各个数组中的最后一个元素：

```
@tailings = popmany ( \@a, \@b, \@c, \@d );

sub popmany {
    my @retlist = ();
    for my $aref (@_) {
        push @retlist, pop @$aref;
    }
    return @retlist;
}
```

你可能会写类似下面的一个函数，来完成集合交集运算，它会返回一个列表，其中包括传入的所有散列中都出现的键：



```

@common = inter( \%foo, \%bar, \%joe );
sub inter {
    my %seen;
    for my $href (@_) {
        while (my $k = each %$href) {
            $seen{$k}++;
        }
    }
    return grep { $seen{$_} == @_ } keys %seen;
}

```

到目前为止，我们只使用了常规的列表返回机制。如果你想传递或返回一个散列会怎么样呢？嗯，如果你只使用一个散列，或者不考虑散列的连接，那么常规的调用机制就足够了，不过开销有些大。

前面已经解释过，如果有：

```
(@a, @b) = func(@c, @d);
```

或者

```
(%a, %b) = func(%c, %d);
```

这会有问题。这个语法不能正确工作。它只会设置@a或%a，而把@b或%b清空。另外，这个函数不会得到两个单独的数组或散列作为参数：与以往一样，它只会得到@\_中的一个长长的列表。

也许你想调整你的函数，让它使用输入和输出的引用。下面这个函数取两个数组引用作为参数，并返回两个数组引用，按其中包含的元素个数排序：

```

($aref, $bref) = func(\@c, \@d);
print "@$aref has more than @$bref\n";
sub func {
    my ($cref, $dref) = @_;
    if (@$cref > @$dref) {
        return ($cref, $dref);
    } else {
        return ($dref, $cref);
    }
}

```

关于如何将文件句柄或目录句柄传入或传出函数，请参见第8章中“句柄引用”和“符号表引用”节。

## 原型

Perl允许你定义你自己的函数，可以像Perl的内置函数一样调用这些自定义函数。请考虑push(@array, \$item)，它必须接收@array的一个引用，而不是@array中包含的列表值，这样才能修改这个数组。通过原型（Prototypes），可以将子例程声明为需要像很多内置

函数一样接收参数，也就是说，对参数的个数和类型有一些约束。我们将它们称为“原型”，但实际上，对于调用上下文来说，它们更像是一些自动模板，而不是C或Java程序员所认为的“原型”。利用这些模板，Perl会自动增加隐式的反斜线或scalar调用，或者增加认为必要的其他调用，以保证与模板一致。例如，如果声明：

```
sub mypush (+@);
```

那么mypush就要像push那样接收参数。为此，编译时必须能看到所调用函数的声明。只有在忽略&字符时，原型才会影响函数调用的解释。换句话说，如果像内置函数一样调用这个函数，它就表现得像一个内置函数。如果像一个老式子例程那样调用，则表现为一个老式的子例程。&会抑制原型检查和相关的上下文影响。

由于原型只在编译时起作用，很自然的，它们对类似\&foo的子例程引用或类似&{\$subref}或\$subref->()等间接子例程调用并没有影响。方法也不受原型影响。这是因为，要调用的具体函数在编译时是不确定的，要依赖于其继承性，而在Perl中这是动态确定的。

由于我们的目的主要是让你学会定义能够像内置函数一样工作的子例程，表7-1给出了一些原型，你可以使用这些原型来模拟相应的内置函数。

表7-1：模拟内置函数的原型

声明方式	调用方式
sub mylink (\$\$)	mylink \$old, \$new
sub myreverse (@)	myreverse \$a,\$b,\$c
sub myjoin (\$@)	myjoin ":"\$a,\$b,\$c
sub mypop (;+)	mypop @array
sub mysplICE (+;\$@\$)	mysplICE @array,@array,0,@pushme
sub mykeys (+)	mykeys %{\$hashref}
sub mypipe (**)	mypipe READHANDLE, WRITEHANDLE
sub myindex (\$;\$)	myindex &getstring, "substr"
	myindex &getstring, "substr", \$start
sub mysyswrite (*;\$;\$)	mysyswrite UTF, \$buf
	mysyswrite UTF, \$buf, length(\$buf)-\$off, \$off
sub myopen (*;\$@)	myopen HANDLE
	myopen HANDLE, \$name
	myopen HANDLE, "- ", @cmd
sub mysin (_)	mysyn \$a
	mysyn
sub mygrep (&@)	mygrep { /foo/ } \$a,\$b,\$c
sub myrand (\$)	myrand 42
sub mytime ()	mytime

反斜线原型字符（上表左列小括号中显示的字符）表示一个实际的参数（右列中给出了具体示例），它必须以该字符开头。就像`keys`的第一个参数必须以`%`或`$`开头一样，`mykeys`的第一个参数也同样必须以`%`或`$`开头。这一点由`+`原型保证，它是`\[@%]`的快捷方式<sup>注3</sup>。

可以使用反斜线组记法`\[]`，指定允许有多个反斜线参数类型。例如：

```
sub myref (\[$@%*&])
```

这表示可以用以上参数调用`myref`，Perl会安排函数接收指定参数的一个引用：

```
myref $var
myref @array
myref %hash
myref &sub
myref *glob
```

分号将必要参数与可选参数分开（在`@`或`%`前面加分号是冗余的，因为列表可以为空）。未加反斜线的原型字符有特殊的含义。如果未加反斜线，`@`或`%`会吞掉其余的所有实参，并强制为列表上下文（这等价于语法描述中的`LIST`）。`$`表示的参数会强制为一个标量上下文。另外`&`要求是一个命名或匿名子例程的引用。

作为原型的最后一个字符，或分号前的字符，可以使用`_`取代`$`。如果没有提供这个参数，会使用当前的`$_`变量。例如：

```
sub mymkdir(_;$) {
    my $dirname = shift();
    my $mask = @_ ? shift() : 0777;
    my $mode = $mask &~ umask();
    ...
}

mkdir($path, 01750);
mkdir($path);
mkdir(); # 传入$_
```

`+`原型是`$`的一个特殊的替代选择，传入一个直接量数组或散列变量时，它相当于`\[@%]`，否则会对参数强制标量上下文。如果函数不仅接收直接量数组（或散列）参数，还可以接收数组或散列的引用，`+`原型就很有用：

```
sub mypush (+@) {
    my $aref = shift;
    die "Not an array or arrayref" unless ref($aref) eq "ARRAY";
    push @$aref, @_;
}
```

---

注3： 这些年散列操作符的原型有一些改变。在v5.8中是`\%`，在v5.12中是`\[@%]`，到了v5.14中则变成`+`。



使用+原型时，函数要检查参数是否是一种可接受的类型（这里我们故意写为不影响对象，否则这会纵容违反对象的封装原则）。

\*允许子例程在该位置接受文件句柄参数，这可以是内置函数所接受的任何文件句柄参数：包括裸名、常量、标量表达式、类型团或类型团的引用。参数值可以作为一个简单标量，或者（对于后两种情况）作为类型团引用由子例程使用。如果你希望把这些参数转换为一个类型团引用，则可以使用`Symbol::qualify_to_ref`，如下所示：

```
use Symbol "qualify_to_ref";

sub myfileno (*) {
    my $fh = qualify_to_ref(shift, caller);
    ...
}
```

注意表中最后3个例子，解析器会特殊对待这些原型。`mygrep`会解析为一个真正的列表操作符，`myrand`解析为真正的一元操作符，其一元操作符优先级与`rand`相同，`mytime`解析为真正的无参数操作符，就像`time`一样。

也就是说，如果有：

```
mytime +2;
```

你会得到`mytime() + 2`而不是`mytime(2)`，如果没有原型或者有一个一元原型，则会解析为`mytime(2)`。

`mygrep`例子还展示了`&`作为第一个参数时会得到特殊处理。正常情况下，`&`原型要求参数类似于`&foo`或`sub{}`。不过，作为第一个参数时，可以省略匿名子例程的`sub`，只在“间接对象”的位置上传入一个裸块（后面没有逗号）。所以`&`原型很棒的一点是，你可以用它生成新语法，只要`&`在第一个参数位置上：

```
sub try (&$) {
    my ($try, $catch) = @_;
    eval { &$try };
    if ($?) {
        local $_ = $@;
        &$catch;
    }
}
sub catch (&) { $_[0] }

try {
    die "phooey";
} # 函数调用还没有结束!
catch {
    /phooey/ && print "unphooey\n";
};
```

这会打印“unphooey”。这里调用`try`时实际上使用了两个参数：匿名函数`{die`

"phooey";}和catch函数的返回值，在这里这个返回值就是它自己的参数——另一个匿名函数的整个代码块。在try中，调用第一个函数参数时，它保护在一个eval块中以捕获可能出现的错误。如果确实出现问题，则会调用第二个函数，并将全局\$\_变量的局部版本设置为所产生的异常<sup>注4</sup>。听上去可能不好理解，你需要先看看第27章的die和eval，然后再学习第8章的匿名函数和闭包。另一方面，如果你很有兴趣，可以看看CPAN上的Try::Tiny模块，它使用这个原型利用try，catch和finally子句实现了结构精巧的异常处理机制。

下面重新实现了grep BLOCK操作符<sup>注5</sup>（当然，内置操作符更高效）：

```
sub mygrep (&@) {
    my $coderef = shift;
    my @result;
    for my $_ (@_) {
        push(@result, $_) if &$coderef;
    }
    return @result;
}
```

有些人更喜欢全字母数字原型。原型中有意没有使用字母数字，这是因为将来可能会增加命名形参（很有可能），那些形参使用字母数字会更合适。当前这种机制的主要目标是允许编写模块的人强制模块用户完成一定程度的编译时检查。

内置函数prototype用于获得用户自定义函数和内置函数的原型，参见第27章。要动态改变一个函数的原型，可以使用标准Scalar::Util模块的set\_prototype函数。例如，如果你希望Unicode::Normalize的NFD和NFC函数有一个原型“\_”，可以这样做：

```
use Unicode::Normalize qw(NFD NFC);

BEGIN {
    use Scalar::Util "set_prototype";
    set_prototype(\&NFD => "_");
    set_prototype(\&NFC => "_");
}
```

## 内联常量函数

如果函数的原型指定为()，则说明它们根本没有参数，这些函数的解析类似于time内置函数。更有意思的是，编译器会把这些函数当作内联的潜在候选对象。在Perl的优化和常量叠算（constant-folding）处理之后，如果函数的结果是一个常量或者是一个没有其他引用的词法作用域标量，就会用这个值取代对这个函数的调用。不过，使用&NAME完成的调用永远不会内联，因为它们不受任何其他原型的影响（参见第29章的constant pragma，其中提供了声明这些常量的一种简便方法）。

---

注4： 没错，关于@\_的可见性还有一些问题没有解决。我们暂且忽略这个问题。

注5： 不可能重新实现grep EXPR 形式。

以下两个计算 $\pi$ 的函数都会被编译器内联：

```
sub pi ()      { 3.14159 }      # 不精确，但接近
sub PI ()      { 4 * atan2(1, 1) } # 非常好
```

实际上，下面的所有函数都会被内联，因为Perl可以在编译时确定一切：

```
sub FLAG_FOO () { 1 << 8 }
sub FLAG_BAR () { 1 << 9 }
sub FLAG_MASK () { FLAG_FOO | FLAG_BAR }

sub OPT_GLARCH () { (0x1B58 & FLAG_MASK) == 0 }
sub GLARCH_VAL () {
    if (OPT_GLARCH) { return 23 }
    else { return 42 }
}

sub N () { int(GLARCH_VAL) / 3 }
BEGIN {
    # 编译器在编译时运行这个代码块
    my $prod = 1;    # 持久存储的私有变量
    for (1 .. N) { $prod *= $_ }
    sub NFACT () { $prod }
}
```

在最后一个例子中，NFACT函数是内联的，因为它有一个void原型，而且它返回的变量未被该函数修改；另外，任何其他函数也不能改变这个变量，因为它在一个词法作用域中。因此编译器会把NFACT调用替换为这个值，由于用BEGIN包围，所以这会在编译时预计算。

如果重新定义一个可以内联的子例程，你就会得到一个强制警告（可以使用这个警告来判断编译器是否已内联一个特定的子例程）。这个警告很重要，不容忽视，因为以前编译的函数调用还会使用函数原来的值。如果需要重新定义这个子例程，要确保它不被内联，可以删除()原型（这会改变调用语义，所以要当心），或者以另外某种方式阻止内联，如：

```
sub not_inlined () {
    return 23 if $$;
}
```

关于程序的编译和执行阶段发生了什么，更多内容请参见第16章。

## 谨慎使用原型

最好在新函数上加原型，而不要改造老函数上的原型。Perl中的原型只是上下文模板，而不是ANSI C原型，所以一定要特别当心有可能会不经意地指定另一个不同的上下文。例如，假设你希望函数只有一个参数，如下：

```
sub func ($) {
    my $n = shift;
    print "you gave me $n\n";
}
```



这会使它成为一个一元操作符（类似于rand内置函数），而且会改变编译器确定函数参数的方式。基于这个新原型，函数认为只有一个标量上下文参数，而不是列表上下文的多个参数。如果调用这个函数时提供一个数组或列表表达式（数组或列表中只包含一个元素），以前这样做是可以的，但是现在你会得到完全不同的结果：

```
func @foo;          # 统计@foo元素个数
func split /:/;     # 统计返回的字段个数
func "a", "b", "c"; # 只传入一个"a"，而丢掉"b"和"c"
func("a", "b", "c"); # 出现编译器错误！
```

你在参数表前面提供了一个隐式的scalar（标量），结果可能让人吃惊不小。尽管老的@foo只包含一个元素，但是并没有将这个数组传入函数。实际上，现在只是把1（@foo中的元素个数）传入了func。再看split，它在标量上下文中调用，这会略过@\_参数表，而只统计参数个数。在第三个例子中，由于func的原型指定它是一个一元操作符，所以只传入了“a”；然后会丢掉func的返回值，因为逗号操作符会继续计算下面两项，并返回“c”。再看最后一个例子，尽管原先这个代码可以编译而且能很好地运行，但现在用户会在编译时得到一个语法错误。

如果要写新的代码，希望一元操作符只取一个标量变量，而不允许老的标量表达式作为参数，可以指定原型，要求它只取一个标量引用（reference）：

```
sub func (\$) {
    my $nref = shift;
    print "you gave me $$nref\n";
}
```

现在，除了以美元符开头的变量外，传入其他参数时编译器都会报错：

```
func @foo;          # 编译器错误，想要$但看到@
func split /:/;     # 编译器错误，想要$但看到函数
func $s;            # 这是可以的——有$符号
func $a[3];         # 这也是可以的
func $h{stuff}[-1]; # 或者甚至这样也可以
func 2+5;           # 标量表达式会导致编译器错误
func ${ \ (2+5) };  # 可以，不过这个大括号是不是很糟糕？
```

如果不小心，就会因为原型遇到麻烦。不过，如果你很谨慎，则可以利用原型做很多很棒的工作。原型的作用很强大，当然，只有适当使用才能得到更好的结果。

## 内置函数的原型

为便于参考，表7-2列出了v5.14中可覆盖的内置函数原型。

表7-2: 内置函数的原型

原型	关键字
()	and, break, continue, dump, endgrent, endhostent, endnetent, endprotoent, endpwent, endservent, fork, getgrent, gethostent, getlogin, getnetent, getppid, getprotoent, getpwent, getservent, or, setgrent, setpwent, time, times, wait, wantarray
(_)	abs, alarm, chr, chroot, cos, exp, fc, hex, int, lc, lcfirst, length, log, oct, ord, quotemeta, readlink, readpipe, ref, rmdir, sin, sqrt, uc, ucfirst
(;\$)	caller, chdir, exit, getpgrp, gmtime, localtime, rand, reset, sleep, srand, umask,
(;*)	close, eof, getc, readline, select, tell, write
(;+)	pop, shift
(@)	chmod, chown, die, kill, reverse, unlink, utime, warn
(_;\$)	mkdir
(;\$ \$)	setpgrp
(\$)	getgrgid, getgrnam, gethostbyname, getnetbyname, getprotobyname, getprotobynumber, getpwnam, getpwuid, sethostent, setnetent, setprotoent, setservent
(*)	closedir, fileno, getpeername, getsockname, lstat, readdir, rewinddir, stat, telldir
(+)	each, keys, values
(\ \$)	lock
(\ %)	dbmclose
(\ [\$@%*])	tied, untie
(\$;\$)	bless, unpack
(*;\$)	binmode
(*;\$@)	open
(+;\$ \$@)	splice
(\$ \$)	atan2, crypt, gethostbyaddr, getnetbyaddr, getpriority, getservbyname, getservbyport, link, msgget, rename, semop, symlink, truncate, waitpid
(\$@)	formline, join, pack, sprintf, syscall
(+@)	push, unshift
(*\$)	bind, connect, flock, listen, opendir, seekdir, shutdown
(**)	accept, pipe
(\$ \$;\$)	index, rindex
(\$ \$;\$ \$)	substr
(*\$;\$ \$)	syswrite
(\ [\$@%*] \$@)	tie
(\$ \$ \$)	msgctl, msgsnd, semget, setpriority, shmctl, shmget, vec
(*\$ \$)	fcntl, getsockopt, ioctl, seek, sysseek

表7-2：内置函数的原型（续）

原型	关键字
(\%\$\$)	dbmopen
(*\$\$;\$)	send, sysopen
(*\\$\$\$;)	read, sysread
(\$\$\$\$)	semctl, shmread, shmwrite
(*\$\$\$)	setsockopt, socket
(*\\$\$\$)	recv
(\$\$\$\$)	msgrcv
(**\$\$\$)	socketpair

## 子例程属性

子例程声明或定义可以关联一个属性列表。如果有这样一个属性列表，会在空白符或冒号边界分解，就好像有一个`use attributes`声明一样。有关的内部细节参见第29章的`attributes pragma`。关于子例程有两个标准属性：`method`和`lvalue`。

### method属性

`method`属性可以单独使用：

```
sub afunc : method { ... }
```

目前，这个属性只有一个作用：就是标识子例程，使它不会触发“`Ambiguous call resolved as CORE::%s`”警告（以后可能会赋予它更多含义）。

属性系统是用户可扩展的，你可以创建自己的属性名。这些新属性必须合法，即必须是简单的标识符名（除“`_`”字符外不能有其他标点符号）。属性后面可以追加一个参数表，目前只会检查参数表的括号是否正确嵌套。

下面给出的几个例子都有正确的语法（尽管这些属性是未知的）：

```
sub fnord (&\%) : switch(10,foo(7,3)) : expensive;
sub plugh () : Ugly('\(") :Bad;
sub xyzzy : _5x5 { ... }
```

下面的例子中语法不正确：

```
sub fnord : switch(10,foo()); # 括号字符串不平衡
sub snoid : Ugly("("); # 括号字符串不平衡
sub xyzzy : 5x5; # "5x5"不是一个合法的标识符
sub plugh : Y2::north; # "Y2::north"不是一个简单标识符
sub snurt : foo + bar; # "+"不是一个冒号或空格
```

属性列表作为一个常量字符串列表传递给代码，代码将这些属性与子例程关联。具体



如何处理（或不处理）还在试验中。关于属性列表及其处理的最新详细信息请查看 *attributes(3)*。

## lvalue属性

可能会从子例程返回一个可修改的标量值，不过前提是必须声明这个子例程返回一个左值（lvalue）：

```
my $val;
sub canmod : lvalue {
    $val;
}
sub nomod {
    $val;
}

canmod() = 5;      # 赋至$val
nomod() = 5;       # 错误
```

对于声明将要返回左值的子例程，如果要向这个子例程传递参数，通常需要加小括号来明确赋给参数的值：

```
canmod $x = 5;      # 先把5赋给$x!
canmod 42 = 5;      # 不能改变一个常量：编译时错误
canmod($x) = 5;     # 这是可以的
canmod(42) = 5;     # 这样也可以
```

如果想简单一些，对于这种特殊情况，即子例程只有一个参数时，可以不加小括号。声明函数时如果指定原型(\$)，函数解析时则有命名一元操作符的优先级。由于命名一元操作符的优先级高于赋值操作符，所以不再需要小括号（这种情况下，是否加小括号完全由代码风格决定）。

还有一种特殊情况：子例程允许0个参数（也就是说，有一个()原型）。由于没有二义性，所以可以放心地写为：

```
canmod = 5;
```

这是可以的，因为任何合法的项都不会以=开头。类似的，如果没有传递任何参数，声明将返回左值的方法调用也可以忽略小括号：

```
$obj->canmod = 5;
```

我们承诺在Perl 5将来的版本中不会改变这两个构造。如果你希望在方法调用中包装对象属性，它们会很方便（这样就能像方法调用一样继承这些属性，但是可以像变量一样访问）。

要确定左值子例程和子例程赋值表达式右边是标量上下文还是列表上下文，可以假设将子

例程调用替换为一个标量。例如，考虑以下调用：

```
data(2,3) = get_data(3,4);
```

这里的两个子例程都在标量上下文中调用，而在下面的代码中：

```
(data(2,3)) = get_data(3,4);
```

以及以下代码中：

```
(data(2),data(3)) = get_data(3,4);
```

所有子例程都在列表上下文中调用。

当前的实现中，不允许从左值子例程直接返回数组和散列。不过完全可以返回数组或散列的一个引用。

由于实践和哲学方面的原因，Perl一直因其扁平的线性数据结构而受到歧视。不过，对于很多问题，这种扁平的数据结构正是你想要的。

假设你想建立一个简单的表格（一个二维数组），显示一个人群的重要统计数据，包括年龄、眼睛颜色和体重。可以先为每个人创建一个数组：

```
@john = (47, "brown", 186);  
@mary = (23, "hazel", 128);  
@bill = (35, "blue", 157);
```

然后构造另一个数组，其中包含之前创建的其他数组的数组名：

```
@vitals = ("john", "mary", "bill");
```

因为John在小镇狂欢了一夜，现在要把他的眼睛颜色改成红色（“red”），我们希望能有一种方法，只给出简单的字符串“john”就能修改@john数组的内容。这正是间接（indirection）的基本问题，不同的语言采用了多种不同的方式来解决这个问题。在C中，最常见的间接方式就是指针，利用指针，一个变量可以包含另一个变量的内存地址。在Perl中，最常见的间接形式是引用（reference）。

## 什么是引用？

在这个例子中，\$vitals[0]的值是“john”。也就是说，它包含一个字符串，这个字符串正好是另一个（全局）变量的名字。我们说第一个变量引用了第二个变量，这种引用称为符号引用（symbolic reference），因为Perl必须在一个符号表中查找@john（可以认为符号引用类似于文件系统中的符号链接）。本章后面还会讨论符号引用。



还有一种引用称为硬引用（hard reference），这也是大多数Perl程序员实现间接机制所用的引用。我们称为硬引用并不是因为它们很难（译者注：“hard”也有难的意思），而是因为它们是真实的、可靠的。如果你愿意，可以把硬引用认为是真实的引用，而符号引用是假引用。这有点像真正的友谊与点头之交的差别。如果我们没有具体指出是哪种引用，那么就是硬引用。图8-1显示了一个名为\$bar的变量引用了标量\$foo的内容，即值“bot”。

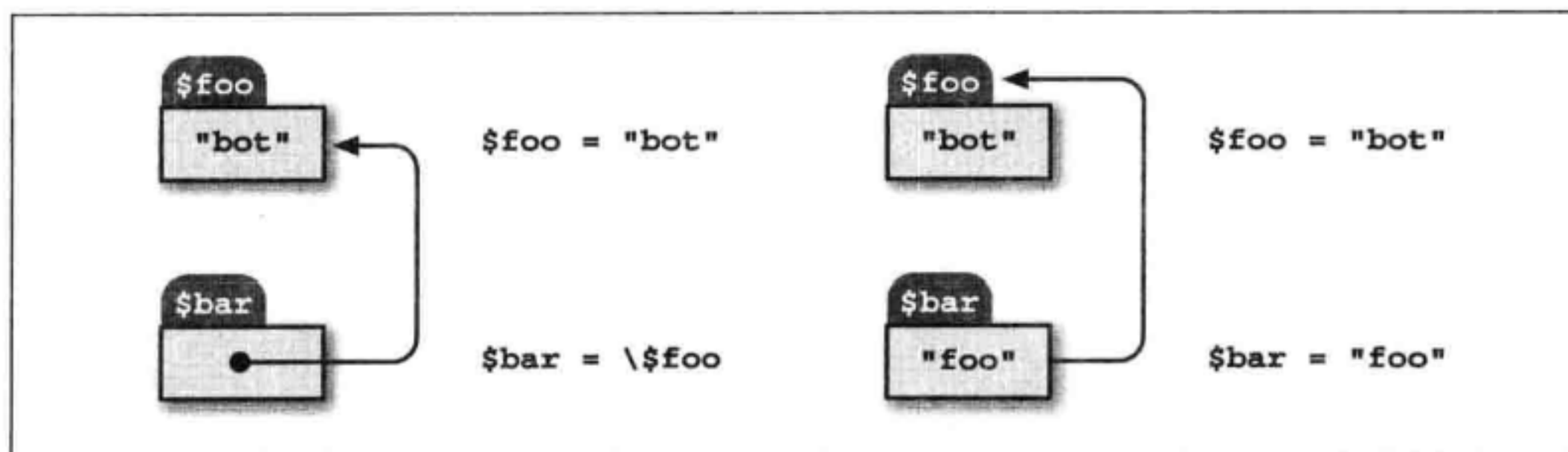


图8-1：硬引用和符号引用

与符号引用不同，真正的引用并不是引用另一个变量的名字（那只是值的容器），而是引用具体的值本身，这是内部的一个数据团。对此并没有一个特别合适的词，不过如果非得给它一个名字，可以称为“指示对象”（referent）。例如，假设你创建了一个词法作用域数组@array的硬引用。即使@array已经出了作用域，这个硬引用以及它引用的指示对象仍继续存在。对于指示对象来说，只有当指向它的所有引用都取消，这个指示对象才会被撤销。

除了引用之外，指示对象并没有自己的名字。换句话说，每个Perl变量名都存在于某个符号表中，包含一个硬引用指向其底层指示对象（否则这个指示对象是没有名字的）。这个指示对象可能很简单，如一个数字或字符串，也可能很复杂，如一个数组或散列。不管怎样，只有一个引用从变量指向相应的值。可以为相同的指示对象创建多个硬引用，不过，如果这样做，变量并不知道（也不关心）这些引用<sup>注1</sup>。

符号引用就是一个字符串，这恰好是包符号表中某个变量的名字。符号引用与字符串并不是截然不同的类型。不过硬引用则完全不同。它是3种基本标量数据类型中的第3种类型，另外两个分别是字符串和数字。硬引用并不知道要引用的变量的名字，实际上，即使变量根本没有名字也是很正常的。这种没有名字的指示对象称为匿名指示对象（anonymous），我们将在这一章后面的“匿名数据”一节讨论这个内容。

要引用一个值，按照这一章的术语来讲，就是要创建这个值的一个硬引用（有一个特殊的操作符可以完成硬引用的创建）。所创建的引用只是一个标量，在我们熟悉的各种上下文

注1：如果你很好奇，可以用Perl提供的Devel::Peek模块确定底层的引用计数（refcount）。

中它与其他标量的行为类似。要对这个标量解引用，是指使用这个引用得到具体的指示对象。引用和解引用只会在调用某种显式机制时才会出现；在Perl 5中不会发生隐式的引用或解引用。嗯，确切地讲，几乎不会发生<sup>注2</sup>。

函数调用可以使用隐式的传引用语义（即按引用传递），前提是它有一个原型对这一点做了声明。如果是这样，函数的调用者不会显式地传递引用，不过在函数中仍然必须显式地解引用。参见第7章中的“原型”一节。诚实地讲，使用某些类型的文件句柄时，实际上在后台也会发生解引用，不过这是为了保持向后兼容，希望对临时用户透明。有两个内置函数**bless**和**lock**，它们分别取一个引用作为参数，不过会隐式地解引用来处理实际引用的值。最后要说明的是，在v5.14版本中，有一些专门处理数组和散列的内置函数<sup>注3</sup>，它们现在可以接受指向正常类型的引用，并根据需要解引用。不过除此以外，基本原则仍然不变，即Perl对于介入你的间接层次并没有兴趣。

引用可以指向任意的数据结构。由于引用是标量，可以把它们存储在数组和散列中，相应地可以构建数组的数组、散列的数组、数组的散列，以及散列和函数的数组等。第9章给出了相应的一些例子。

不过，要记住，Perl数组和散列在内部都是一维的。也就是说，它们的元素只能包含标量值（字符串、数字和引用）。我们谈到“数组的数组”时，实际上是指“包含数组引用的数组”，而谈到“函数的散列”时，实际上是指“包含子例程引用的散列”。不过，由于引用是Perl中实现这些结构的唯一途径，所以尽管这些比较简短的说法不太准确，但还不算太离谱，不能算是错误。因此，不要鄙视这些说法，除非你很挑剔。

## 创建引用

创建引用有很多方法，大多会在这里介绍，然后再解释如何使用（解引用）得到的引用。

## 反斜线操作符

可以用一个反斜线创建任何命名变量或子例程的引用（还可以把反斜线操作符用于匿名标量值，如7或"camel"，不过通常并不需要这么做）。这个操作符有些像C中的&（取地址）操作符，至少初看是这样。

下面给出几个例子：

```
$scalarref = \ $foo;  
$constref = \186_282.42;  
$arrayref = \@ARGV;
```

---

注2：在Perl 6中则刚好相反，几乎都是隐式的，这会让你很困惑。

注3：keys, values, each, pop, push, shift, unshift和splice。



```
$hashref = \%ENV;
$coderef = \&handler;
$globref = \*STDOUT;
```

反斜线操作符的作用不只是生成一个引用。如果应用于一个列表，它会生成整个引用列表。有关的详细内容参见后面的“使用硬引用的其他技巧”一节。

## 匿名数据

在前面给出的例子中，反斜线操作符只是建立了变量名中已包含的引用的一个副本，只有一个例外。186\_282.42并非由一个命名变量引用，它只是一个值。这就是我们之前提到的匿名指示对象。匿名指示对象只能通过引用来访问。这个指示对象恰好是一个数字，不过也可以创建匿名数组、散列和子例程。

## 匿名数组生成器

可以用中括号创建一个匿名数组的引用：

```
$arrayref = [1, 2, ["a", "b", "c", "d"]];
```

这里我们构造了一个包含3个元素的匿名数组，最后一个元素是一个引用，它指向另一个包含4个元素的匿名数组（如图8-2所示）。（可以使用多维语法来访问，例如\$arrayref->[2][1]的值为“b”，多维语法的有关内容将在后面介绍）。

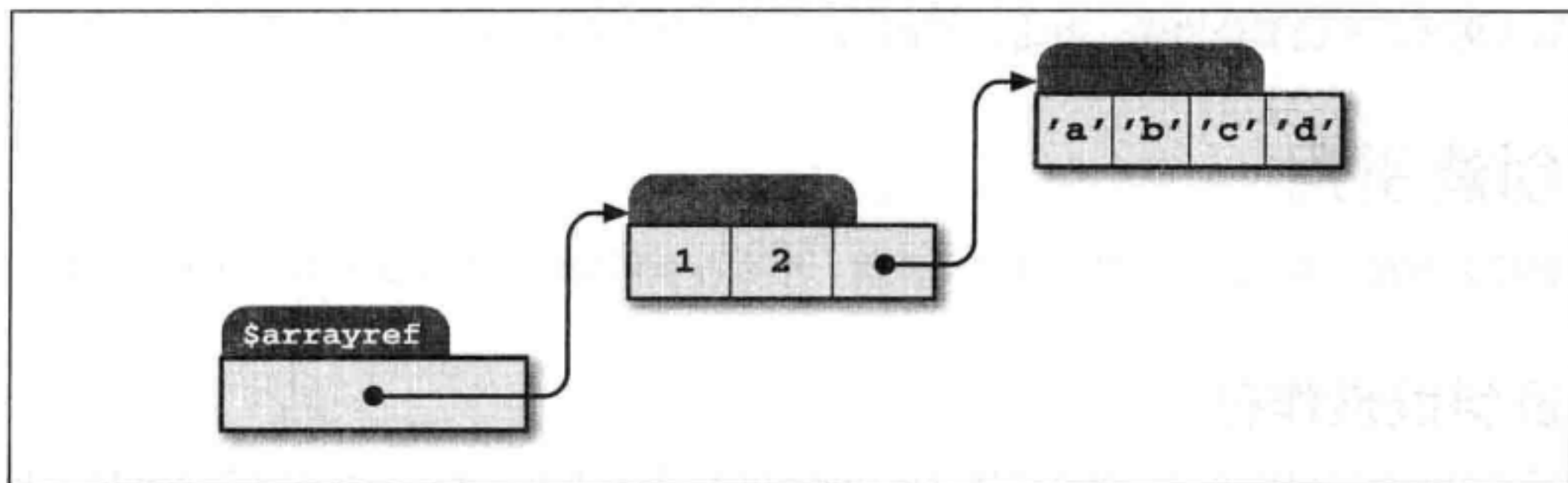


图8-2：一个数组引用，其中第3个元素本身也是一个数组引用

现在我们得到一种方法来表示本章开头给出的表格：

```
$table = [ [ "john", 47, "brown", 186],
            [ "mary", 23, "hazel", 128],
            [ "bill", 35, "blue", 157] ];
```

只有当Perl解析器希望得到表达式中的一项时，中括号才会像这样工作。不要把它们与表达式中的中括号（如\$array[6]）混淆，不过与数组的这种关联是我们有意保留的。在引号字符串中，中括号不生成匿名数组；它们会成为字符串中的直接量字符（中括号在字符



串中仍可以用于指定下标，否则你将无法打印类似"VAL\$array[6]\n"的字符串值。说实在的，实际上可以把匿名数组生成器放在字符串中，不过只有当嵌套在一个更大的表达式中而且这个表达式要完成内插时才有效。本章后面还会介绍这个很酷的特性，因为它不仅涉及引用，还涉及解引用）。

## 匿名散列生成器

可以用大括号创建匿名散列的引用：

```
$hashref = {  
    "Adam" => "Eve",  
    "Clyde" => $bonnie,  
    "Antony" => "Cleo" . "patra",  
};
```

对于散列的值（而不是键），可以自由地混合其他匿名数组、散列和子例程生成器，生成你需要的任意复杂的结构。

现在我们有了另一种方法来表示本章开头给出的表格：

```
$table = {  
    "john" => [ 47, "brown", 186 ],  
    "mary" => [ 23, "hazel", 128 ],  
    "bill" => [ 35, "blue", 157 ],  
};
```

这是一个数组的散列。选择最佳的数据结构是一个很棘手的事情，下一章就要解决这个问题。不过，为了好玩，甚至可以使用一个散列的散列来创建我们的表格：

```
$table = {  
    "john" => { age => 47,  
                eyes => "brown",  
                weight => 186,  
            },  
    "mary" => { age => 23,  
                eyes => "hazel",  
                weight => 128,  
            },  
    "bill" => { age => 35,  
                eyes => "blue",  
                weight => 157,  
            },  
};
```

与中括号类似，仅在Perl解析器希望得到表达式中的一项时大括号才会这样做。不要把它们与表达式中的大括号混淆，如\$hash{key}。不过（同样的）这种用法与散列的关联也是我们有意保留的。在字符串中使用大括号也存在同样的问题。不过还有一个问题是中括号所没有的。由于大括号还用于很多其他构造（包括代码块），有时可能必须在语句开头消除大括号的歧义，在前面加一个+或者return，这样Perl就能知道这个开始大括号并不是要

开始一个代码块。例如，如果你希望函数建立一个新散列，并返回这个散列的引用，可以有以下选择：

```
sub hashem {      { @_ } } # 错误， 这会返回@_  
sub hashem {      +{ @_ } } # 正确  
sub hashem { return { @_ } } # 正确
```

## 匿名子例程生成器

可以使用sub但不加子例程名来创建一个匿名子例程的引用：

```
$coderef = sub { print "Boink!\n" }; # 现在&$coderef会打印"Boink!"
```

注意这里的分号，必须有这个分号才能结束表达式（声明和定义命名子例程更常用的方式是sub NAME {}, 后面则不要求有分号）。没有名字的sub {}与其说是一个声明，不如说是一个操作符，类似于do {}或eval {}, 只不过其中的代码不会立即执行。实际上，它只生成代码的一个引用，在这个例子中，这个引用存储在\$coderef中。不过，不论将上面这行代码执行多少次，\$coderef仍指向同一个匿名子例程<sup>注4</sup>。

## 对象构造函数

子例程还可以返回引用。听起来好像很平常，不过有时你要使用子例程来创建一个引用而不是由你自己创建引用。具体来讲，一些特殊的子例程会创建对象并返回对象的引用，这些子例程称为构造函数（constructors）。对象只是一种特殊类型的引用，它知道与哪个类关联，而构造函数知道如何创建这个关联。为此，要用bless操作符把一个普通的指示对象（referent）转变为一个对象（object），所以我们也把对象称为一个“被祝福”（blessed）的引用。这与信仰无关；因为类相当于一个用户定义的类型，祝福一个指示对象只是让它除了有内置类型外还有一个用户自定义的类型。构造函数通常名为new，特别是C++和Java程序员很喜欢这么做。不过在Perl中，构造函数可以有任意的名字。

构造函数可以采用以下任何一种方法调用：

```
$objref = Doggie::->new(Tail => "short", Ears => "long"); #1  
$objref = new Doggie:: Tail => "short", Ears => "long";   #2  
$objref = Doggie->new(Tail => "short", Ears => "long");   #3  
$objref = new Doggie Tail => "short", Ears => "long";     #4
```

第一个和第二个调用是一样的。它们都调用一个名为new的函数，这个函数由Doggie模块提供。第三个和第四个调用与前面两个相同，不过稍稍含糊一点：如果你定义了自己的Doggie子例程，解析器就会有些糊涂（正是因为这个原因，人们通常对子例程使用小写的名字，而模块使用大写的名字）。如果你定义了自己的新子例程，不过其中没有用require

---

注4： 不过，尽管只有一个匿名子例程，但子例程使用的词法作用域变量可能有多个副本，这取决于子例程引用何时生成。这些内容将在后面的“闭包”一节中讨论。

或use加载Doggie模块（require和use都有声明这个模块的作用），那么第四个调用也会不知所措。如果你想使用第4种调用方法，一定要声明你的模块（另外要当心自定义的Doggie子例程）。

关于Perl对象的讨论请见第12章。

## 句柄引用

文件句柄或目录句柄的引用可以通过引用同名的类型团来创建：

```
splutter(\*STDOUT);

sub splutter {
    my $fh = shift;
    say $fh "her um well a hmmm";
}

$rec = get_rec(\*STDIN);
sub get_rec {
    my $fh = shift;
    return scalar <$fh>;
}
```

如果传递文件句柄，还可以使用裸类型团来做到：在上面的例子中，可以使用\*STDOUT或\*STDIN而不是\\*STDOUT和\\*STDIN。

尽管通常可以交替使用类型团和类型团引用，不过有些情况下这是不允许的。简单的类型团不能通过bless祝福为对象，而且类型团引用不能超出局部化类型团的作用域传递。

生成新的文件句柄时，较早的代码通常会如下打开一个文件列表：

```
for $file (@names) {
    local *FH;
    open(*FH, $file) || next;
    $handle{$file} = *FH;
}
```

这仍然是可行的，不过通常更好的办法是让一个未定义的变量自动生成（autovivify）为一个匿名类型团：

```
for $file (@names) {
    my $fh;
    open($fh, $file) || next;
    $handle{$file} = $fh;
}
```

如果变量包含一个文件句柄而不是裸字句柄，你就有了一个间接文件句柄。使用字符串、类型团、类型团引用还是更奇怪的I/O对象都不重要。关键是要使用一个标量，它以某种



方式解释为一个文件句柄。对于大多数用途来说，几乎可以无差别地使用类型团或类型团引用。前面已经指出，这里会有一些隐式的解引用魔法。

## 符号表引用

在一些特别的情况下，写程序时你可能不知道需要什么类型的引用。引用可以使用一种特殊的语法来创建，这称为`*foo{THING}`语法。`*foo{THING}`会返回`*foo`中`THING`槽的引用，这是一个符号表记录，其中包含`$foo`、`@foo`、`%foo`等的值。

```
$scalarref = *foo{SCALAR};    # 等同于\ $foo
$arrayref  = *ARGV{ARRAY};    # 等同于\ @ARGV
$hashref   = *ENV{HASH};      # 等同于\ %ENV
$coderef   = *handler{CODE};  # 等同于\ &handler
$globref   = *foo{GLOB};      # 等同于\ *foo
$ioref     = *STDIN{IO};       # ...
$formatref = *foo{FORMAT};    # ...
```

除了最后两个，以上代码都很好理解。`*foo{FORMAT}`可以得到使用`format`语句声明的对象。利用它们做不了太多有趣的工作。

另一方面，`*STDIN{IO}`会得到类型团包含的具体的内部`IO::Handle`对象；也就是说，类型团中各个I/O函数真正感兴趣的部分。为了与老版本的Perl兼容，`*foo{FILEHANDLE}`曾经是较新的`*foo{IO}`记法的同义词，不过这种用法现在已经过时。

理论上讲，只要能使用`*HANDLE`或`\*HANDLE`的地方都可以使用`*HANDLE{IO}`，如向子例程传入或从子例程传出句柄，或者将它们存储在更大的数据结构中（在实际中，这样互换使用还有一些问题需要解决）。它们的好处是只访问你想要的实际I/O对象，而不是整个类型团，所以不用担心通过类型团赋值会遇到太大的风险（不过，如果你总是为标量变量而不是类型团赋值，那肯定没问题）。缺点是目前还无法自动生成I/O对象<sup>注5</sup>：

```
splutter(*STDOUT); splutter(*STDOUT{IO});

sub splutter {
    my $fh = shift; print $fh "her um well a hmmm\n";
}
```

两个`splutter`调用都会打印“her um well a hmmm”。

如果编译器还没有看到特定的`THING`，`*foo{THING}`会返回`undef`，除非`THING`是`SCALAR`。`*foo{SCALAR}`会返回一个匿名标量的引用（即使编译器还没有看到`$foo`）。Perl总是为类型团增加一个标量，把这作为一个优化措施，以减少别处的一些代码。不过不要过分依赖于这一点，将来的版本中可能不是这样。

---

注5：目前，`open my $fh`会自动生成一个类型团而不是一个`IO::Handle`对象，不过将来我们可能会修正这个问题，所以你不要过分依赖于目前`open`会自动生成类型团这一特性。

## 隐式创建引用

前面我们简单地提到了自动生成（autovivifying），这是创建引用的最后一种方法，尽管实际上这并不算是一个方法。如果在一个左值上下文中（假设其存在）解引用，就会隐式地创建适当类型的引用。这很有用，也正是你希望的。这个内容将在这一章后面详细介绍，我们会讨论如何将到目前为止创建的所有引用解引用。嘿，下面就来讨论这个内容。

## 使用硬引用

创建引用有很多不同的方法，同样的，使用引用或解引用也有很多方法。不过有一个最重要的原则：Perl没有隐式引用或隐式解引用<sup>注6</sup>。一个标量包含一个引用时，总是表现为一个简单的标量。它不会魔法般地变成一个数组、散列或子例程；要想让它转变，必须通过解引用显式地要求它变成数组、散列或子例程。

## 使用变量作为变量名

遇到一个类似\$foo的标量时，你可能会想到“foo的标量值”。也就是说，符号表中有一个foo记录，\$字符（称为印记）是查看其标量值的一种方法。如果foo记录中是一个引用，可以附加另一个印记来查找这个引用内部的值（对\$foo解引用）。或者用另一种方式查看，可以把\$foo中的直接量foo替换为指向具体指示对象的标量变量。这对于任何变量类型都是可行的，所以\$\$foo就是\$foo指示的标量值，不仅如此，@\$bar是\$bar指示的数组值，%\$glarch是\$glarch指示的散列值，依此类推。总之，你可以在任何简单的标量变量前面加上一个额外的印记将它解引用：

```
$foo      = "three humps";
$scalarref = \ $foo;      # $scalarref现在是$foo的一个引用
$camel_model = $$scalarref; # $camel_model现在是"three humps"
```

下面再给出另外一些解引用的例子：

```
$bar = $$scalarref;

push(@$arrayref, $filename);
$$arrayref[0] = "January";          # 设置@$arrayref的第一个元素
@$arrayref[4..6] = qw/May June July/; # 设置@$arrayref的多个元素
%$hashref = (KEY => "RING", BIRD => "SING"); # 初始化整个散列
$$hashref{KEY} = "VALUE";           # 设置一个键/值对
@$hashref{"KEY1", "KEY2"} = ("VAL1", "VAL2"); # 设置另外两个键/值对

&$coderef(1,2,3);

say $handlerref "output";
```

---

注6： 我们已经承认这里撒了一个小谎。下回再不这样了。



这种形式的解引用只能使用一个简单的标量变量（没有下标的变量）。也就是说，解引用发生在数组或散列查找之前（或者比数组或散列查找绑定更紧密）。下面使用一些大括号更明确地表述我们的意思：类似`$$arrayref[0]`的表达式等价于`${$arrayref}[0]`，表示`$arrayref`引用的数组中的第一个元素。这与`${$arrayref[0]}`并不相同，`${$arrayref[0]}`是对名为`@arrayref`的数组（可能不存在）中的第一个元素解引用。类似的，`$$hashref{KEY}`等同于`${$hashref}{KEY}`，这与`${$hashref{KEY}}`完全不同，`${$hashref{KEY}}`会对名为`%hashref`的散列（可能不存在）中的一项解引用。必须了解这些，否则你会很麻烦。

通过连接适当的印记，可以得到多层引用和解引用。下面的代码会打印“howdy”：

```
$refrefref = \\\\"howdy";
print $$$$refrefref;
```

可以认为美元符是从右向左处理。但是这个链必须从一个简单的没有下标的标量变量开始。不过，还有一种做法更有意思，之前我们已经偷偷用过，下一节将详细解释。

## 使用BLOCK作为变量名

不仅可以对一个简单的变量名解引用，还可以对一个BLOCK的内容解引用。在任何地方只要能够放入一个字母数字标识符作为变量或子例程名的一部分，都可以用一个返回正确类型引用的BLOCK来替换这个标识符。换句话说，前面的例子都可以如下消除二义性：

```
$bar = ${$scalarref};
push(@{$arrayref}, $filename);
${$arrayref}[0] = "January";
@{$arrayref}[4..6] = qw/May June July/;
${$hashref}{KEY} = "VALUE";
@{$hashref}{KEY1,KEY2} = ("VAL1","VAL2");
&{$coderef}(1,2,3);
```

当然也可以这样：

```
$refrefref = \\\\"howdy";
print ${${${$refrefref}}};
```

必须承认，对于这么简单的情况使用大括号有点傻，不过BLOCK可以包含任意的表达式。具体地，它可以包含有下标的表达式。在下面的例子中，假设`$dispatch{$index}`包含一个子例程的引用（有时称为“coderef”）。这个例子调用这个子例程时提供了3个参数：

```
&{ $dispatch{$index} }(1, 2, 3);
```

在这里，BLOCK是必要的。如果没有最外层的一对大括号，Perl就会把`$dispatch`当作子例程引用（coderef）而不是`$dispatch{$index}`。



## 使用箭头操作符

对于数组、散列或子例程的引用，还有第三种解引用的方法，需要使用`->`中缀操作符。利用这种形式的语法糖，可以更容易地得到单个数组或散列元素，或者间接地调用一个子例程。

解引用的类型由右操作数确定，也就是说，由紧跟在箭头后面的项确定。如果箭头后面是一个中括号或大括号，左操作数会分别看作是一个数组或散列的引用，将以右边表达式作为下标。如果下一项是一个左小括号，左操作数将看作是一个子例程的引用，将用右边小括号中提供的参数来调用这个子例程。

下面给出了几组代码，每一组中包含的3行代码都是等效的，分别对应我们介绍过的3种记法。（这里插入了一些空格，将等效的元素对齐）。

```
$ $arrayref [2] = "Dorian";           #1
${ $arrayref }[2] = "Dorian";         #2
$arrayref->[2] = "Dorian";             #3

$ $hashref {KEY} = "F#major";          #1
${ $hashref }{KEY} = "F#major";        #2
$hashref->{KEY} = "F#major";           #3

& $coderef (Presto => 192);             #1
&{ $coderef }(Presto => 192);           #2
$coderef->(Presto => 192);               #3
```

可以看到，在每一组中，第三种记法都没有最前面的印记。这个印记要由Perl来猜，正是因为这个原因，不能用这种记法对整个数组、整个散列或数组和散列片段解引用。不过，只要坚持使用标量值，可以在`->`左边使用任何表达式，这也包括可以是另一个解引用，因为多个箭头会从左向右结合：

```
print $array[3]->{"English"}->[0];
```

从这个表达式可以推测出，`@array`的第4个元素可能是一个散列引用，而这个散列中“English”项的值可能是一个数组引用。

需要注意，`$array[3]`和`$array->[3]`并不相同。前者是指`@array`的第4个元素，而后者则是某个（可能匿名）数组中的第4个元素，这个数组的引用包含在`$array`中。

假设现在`$array[3]`未定义。下面的语句仍是合法的：

```
$array[3]->{"English"}->[0] = "January";
```

这正是之前提到的一种情况，即用作左值时（也就是说，为它赋值时），引用会自然存在（或“自动生成”）。如果`$array[3]`未定义，它会自动定义为一个散列引用，这样就可以在其中为`$array[3]->{"English"}`设置一个值。一旦完成，`$array[3]->{"English"}`会自

动定义为一个数组引用，从而可以在这个数组中为第一个元素赋某个值。注意右值稍有不同：`print $array[3]->{"English"}->[0]`只定义`$array[3]`和`$array[3]->{"English"}`，而没有定义`$array[3]->{"English"}->[0]`，因为最后一个元素不是一个左值（在一个右值上下文中定义前两个引用，这可以认为是一个bug。将来我们可能会修正这个bug）。

中括号或大括号之间的箭头是可选的，或者一个结束中括号或大括号与一个间接函数调用的小括号之间的箭头也是可选的。所以可以把前面的代码压缩为：

```
$dispatch{$index}(1, 2, 3);  
$array[3>{"English"}[0] = "January";
```

对于普通的数组，这会提供多维数组，与C的数组类似：

```
$answer[$x][$y][$z] += 42;
```

嗯，没错，并不完全像C的数组。一方面，C不知道如何根据需要扩展数组，而Perl知道。另外，这两个语言（C和Perl）中有些构造尽管很类似，但是会用不同的方式解析。在Perl中，下面这两个语句会完成同样的工作：

```
$listref->[2][2] = "hello";    # 很简洁  
$$listref[2][2] = "hello";    # 有些混乱
```

其中第二个语句可能让C程序员很不适应，他们习惯于使用`*a[i]`来表示“a中第i个元素指向的内容”。不过，在Perl中，5个字符（`$ @ * % &`）实际上比中括号或大括号绑定更紧密<sup>注7</sup>。因此，作为数组引用的是`$$listref`而不是`$listref[2]`。如果想采用C的方式，则必须写为`${$listref[2]}`强制`$listref[2]`先计算，然后才应用前面的`$`解引用操作符，或者必须使用`->`记法：

```
$listref[2]->[$greeting] = "hello";
```

## 使用对象方法

如果引用恰好是一个指向对象的引用，定义这个对象的类可能提供了一些访问对象内部的方法，如果你只是使用这个类（而不是实现这个类），通常就必须使用这些方法。换句话说，要规矩一些，不要把对象当作普通的引用，尽管如果你确实非常需要这么做Perl也是允许的。Perl并没有强制必须保证封装。我们不是极权主义者，不过确实希望有基本的礼貌。

作为这种礼貌的回报，你能够得到对象与数据结构之间的完全正交。需要时，任何数据结构都可以作为对象，如果你不希望这样，数据结构也可以不作为对象。

---

注7： 不过并不是因为操作符优先级。从这个意义上讲，Perl中的印记并不是操作符。Perl的文法只允许在第一个印记后面跟一个简单的变量或代码块，而不允许更复杂的东西。



## 伪散列

伪散列（pseudohash）是一种将数组看作是散列的方法，可以模拟一个有序散列。作为一个处于试验阶段的特性，事实表明伪散列的想法并不好，所以v5.10中去除了这个特性，不过有些人还在使用较早的版本，所以这里我们简单提一下，但你最好不要使用。如果你要用到伪散列，应当使用fields模块的phash和new函数。

v5.10以后fields::phash接口不再可用，不过fields::new还可以使用。但你还是应该考虑使用标准Hash::Util模块中严格的散列。

## 关于硬引用的其他技巧

前面已经提到，反斜线操作符通常应用于单个指示对象来生成单个引用，不过并不一定必须如此。应用在一个指示对象列表上时，它会生成相应引用的一个列表。下面例子中的第二行代码与第一行的工作完全相同，因为反斜线会自动地分布到整个列表中：

```
@reflist = (\$s, \@a, \%h, \&f);      # 包含4个引用的列表
@reflist = \($s, @a %h, &f);          # 作用相同
```

如果带小括号的列表是一个数组或散列，它的所有值会内插，并返回各个值的引用：

```
@reflist = \(@x);                      # 内插数组，然后返回引用
@reflist = map { \$_ } @x;             # 作用相同
```

有内部小括号时也会这样：

```
@reflist = \(@x, (@y));                # 不过只有一个集合扩展
@reflist = (\@x, map { \$_ } @y);      # 作用相同
```

如果尝试对一个散列做这个处理，结果会包含值的引用（如你所料），不仅如此，还会包含键副本的引用（这可能是你没有想到的）。

由于数组和散列片段实际上就是列表，所以可以对数组和散列的片段加反斜线来得到一个引用列表。下面3行代码的作用相同：

```
@envrefs = \@ENV{"HOME", "TERM"};      # 对片段加反斜线
@envrefs = \($ENV{HOME}, $ENV{TERM});  # 对列表加反斜线
@envrefs = ( \ $ENV{HOME}, \ $ENV{TERM} ); # 包含两个引用的列表
```

由于函数可以返回列表，所以你可以对列表应用一个反斜线。如果要调用多个函数，首先将各个函数的返回值内插到一个更大的列表，然后对整个列表加反斜线：

```
@reflist = \fx();
@reflist = map { \$_ } fx();            # 作用相同

@reflist = \(\ fx(), fy(), fz() );
@reflist = ( \fx(), \fy(), \fz() );    # 作用相同
@reflist = map { \$_ } fx(), fy(), fz(); # 作用相同
```



反斜线操作符总会为其操作数提供列表上下文，所以这些函数都会在列表上下文中调用。如果反斜线本身在标量上下文中，最后你会得到函数返回的列表中最后一个值的引用：

```
@reflist = \localtime();      # 9个时间元素的引用
$lastref = \localtime();      # 引用指示是否为日光节省时间
```

从这个方面来看，反斜线就类似于命名的Perl列表操作符，如`print`、`reverse`和`sort`，不论左边怎样，都会为右边提供列表上下文。与命名列表操作符一样，可以使用一个显式的`scalar`来强制后面进入标量上下文：

```
$dateref = \scalar localtime(); # \"Tue Oct 18 07:23:50 2011"
```

可以用`ref`操作符来确定一个引用指示什么。可以把`ref`看作是一个“`typeof`”操作符，如果其参数是一个引用，则返回`true`，否则返回`false`。返回的值取决于所引用对象的类型。内置类型包括`SCALAR`、`ARRAY`、`HASH`、`CODE`、`GLOB`、`REF`、`VSTRING`、`IO`、`LVALUE`、`FORMAT`和`REGEXP`，另外还有类`version`、`Regexp`和`IO::Handle`。这里我们使用`ref`操作符来检查子例程参数：

```
sub sum {
    my $arrayref = shift;
    warn "Not an array reference" if ref($arrayref) ne "ARRAY";
    return eval join("+", @$arrayref);
}

say sum([1..100]); # 5050, 利用欧拉的巧妙算法
```

如果在一个字符串上下文中使用硬引用，它会转换为一个包含有类型和地址的字符串：`SCALAR(0x1fc0e)`。不能完成反向转换，因为在串化过程中会丢失引用计数信息。另外还有一个原因，允许程序访问一个用任意字符串命名的内存地址会非常危险。

可以使用`bless`操作符将指示对象与一个作为对象类的包关联起来。如果这样做，`ref`会返回类名而不是内部类型。字符串上下文中使用的对象引用会返回一个字符串，其中包含外部和内部类型以及内存地址：`MyType=HASH(0x20d10)`或`IO::Handle=IO(0x186904)`。关于对象的更多详细内容请参见第12章。

由于解引用的方法总是要指示需要寻找哪种指示对象，所以与引用类似，可以用同样的方式使用类型团，尽管类型团包含不同类型的多个指示对象。所以`${*main::foo}`和`${\ $main::foo}`都访问同一个标量变量，不过后者更高效一些。

下面的技巧可以将一个子例程调用的返回值内插到一个字符串：

```
say "My sub returned @{[ mysub(1,2,3) ]} that time.";
```

它的工作过程如下。编译时，在双引号字符串中看到`@{...}`时，会把它解析为一个返回引用的代码块。这个代码块中有一个中括号，它会创建大括号内一个匿名数组的引用。所

以在运行时，`mysub(1,2,3)`会在列表上下文中调用，结果将加载到一个匿名数组中，再在代码块中返回这个匿名数组的引用。然后立即通过外围的`@{...}`对数组引用解引用，将数组值内插到双引号字符串中，就像一个普通的数组一样。这种解释适用于任意的表达式，如：

```
say "We need @{ [$n + 5] } widgets!";
```

不过要当心：中括号会为其表达式提供列表上下文。在这个例子中并没有影响，不过对之前的`mysub`调用确实有影响。如果有影响，可以使用一个显式的`scalar`来强制进入标量上下文：

```
say "mysub returns @{ [scalar mysub(1,2,3)] } now.";
```

## 闭包

之前我们谈到用一个没有名字的`sub{}`创建匿名子例程。可以认为这些子例程是运行时定义的，这说明它们有一个生成时间，还有一个定义位置。创建子例程时，作用域中可能有一些变量，而在调用这个子例程时作用域中可能是另外一些变量。

先不考虑子例程，下面来看一个指向词法作用域变量的引用：

```
{
  my $critter = "camel";
  $critterref = \ $critter;
}
```

`$$critterref`的值保持为“camel”，尽管`$critter`会在结束大括号之后消失。不过`$critterref`也可以指示一个指向`$critter`的子例程：

```
{
  my $critter = "camel";
  $critterref = sub { return $critter };
}
```

这就是一个闭包（closure），这个概念来自于LISP和Scheme的函数式编程<sup>注8</sup>。它表示某个特定时刻在一个特定的词法作用域中定义一个匿名函数时，它会假装在这个作用域中运行，尽管之后可能在这个作用域之外调用（纯粹主义者可能会说这不是假装，它确实会在那个作用域中运行）。

换句话说，每次你都会得到词法作用域变量的相同副本，即使之前或之后可能为这个闭包的其他实例创建了该词法作用域变量的其他实例。这样就提供了一种方法，可以在定义子例程时设置子例程中使用的值，而不只是调用子例程时才能设置。

---

注8： 在这里，“函数式”（functional）一词不能理解为“功能失调”（dysfunctional）的反义词。

还可以认为闭包是一种写子例程模板的方法，而无需使用`eval`。词法作用域变量将作为填入这个模板的参数，这对于建立以后运行的小段代码很有用。在基于事件的编程中，这通常称为回调（callback），你可能把一小段代码与某个按键、鼠标单击、窗口显示等事件相关联。用作回调时，闭包会像你期望的那样工作，即使你根本不了解函数式编程也没有关系（要注意，这种闭包概念只适用于`my`变量。全局变量仍像往常一样工作，因为它们不会像词法作用域变量那样创建或撤销）。

闭包还有一种用法，可以在函数生成器（generators）中使用；所谓函数生成器也是函数，它们会创建并返回全新的函数。下面是用闭包实现的一个函数生成器的例子：

```
sub make_saying {
    my $salute = shift;
    my $newfunc = sub {
        my $target = shift;
        say "$salute, $target!";
    };
    return $newfunc;          # 返回一个闭包
}

$f = make_saying("Howdy");    # 创建一个闭包
$g = make_saying("Greetings"); # 创建另一个闭包

# 经过一段时间...

$f->("world");
$g->("earthlings");
```

这会打印以下结果：

```
Howdy, world!
Greetings, earthlings!
```

要特别注意`$salute`仍指示传入`make_saying`的实际值，尽管在匿名子例程运行时`my $salute`已经出了作用域。这正是闭包的作用。由于`$f`和`$g`都包含函数的引用，在调用时，它们仍需要访问`$salute`的不同版本，所以这些版本会自动保留。如果现在重写`$f`，它的`$salute`版本会自动消失（只有当你不再查看变量时，Perl才会将变量清除）。

Perl没有提供对象方法的引用（见第12章的介绍），不过使用闭包可以得到一种类似的效果。假设你希望引用不仅指示方法所表示的子例程，还希望指示其调用子例程（即调用这个子例程时，它会在一个特定对象上调用该方法）。可以很方便地把对象和方法记作绑定在闭包中的词法作用域变量：

```
sub get_method_ref {
    my ($self, $methodname) = @_;
    my $methref = sub {
        # 下面的@_ 与上面的不同！
        return $self->$methodname(@_);
    };
}
```



```

    return $methref;
}

my $dog = new Doggie::
    Name => "Lucky",
    Legs => 3,
    Tail => "clipped";

our $wagger = get_method_ref($dog, "wag");
$wagger->("tail");      # 调用$dog->wag("tail").

```

现在可以让Lucky摇晃尾巴的其余部分，不仅如此，甚至一旦词法作用域变量\$dog出了作用域，就不能再看到Lucky，全局变量\$wagger仍能让它摇尾巴，而不论它在哪里。

## 闭包作为函数模板

使用一个闭包作为函数模板，你可以生成多个有类似动作的函数。假设你希望一组函数生成有不同颜色的HTML字体：

```
print "Be ", red("careful"), "with that ", green("light"), "!!!";
```

red和green函数很类似。我们喜欢为函数命名，不过闭包没有名字，因为它们只是要达到某种目的的匿名子例程。为了避开这个问题，我们会实现一个很巧妙的技巧来对匿名子例程命名。通过把一个子例程引用（coderef）赋至一个类型团（这个类型团有你需要的名字），可以为这个coderef绑定一个已有的名字（参见第10章“符号表”一节）。在这里，我们把它绑定到两个不同的名字，一个大写，另一个小写：

```

@colors = qw(red blue green yellow orange purple violet);
for my $name (@colors) {
    no strict "refs";      # 允许符号引用
    *$name = *{uc $name} = sub { "<FONT COLOR='$name'>";@_</FONT>" };
}

```

现在可以调用名为red、RED、blue、BLUE等的函数，从而调用适当的闭包。这个技术可以减少编译时间，并节省内存，而且不容易出错，因为这样会在编译时完成语法检查。有一点很重要，匿名子例程中的所有变量必须是词法作用域变量，这样才能创建闭包。这也是前面使用my的原因。

为闭包指定原型通常没有意义，不过也有很少的几个例外，这里就是一种例外情况。如果想对这些函数的参数强制标量上下文（对这个例子来说可能不是明智的想法），你可能会写为：

```
*$name = sub ($) { "<FONT COLOR='$name'>$_[0]</FONT>" };
```

这已经够好了。不过，由于原型检查在编译时进行，所以上运行时赋值发生得太晚，所以没有太大用处。可以修正这个问题，把整个赋值循环放在一个BEGIN块中，要求它在编

译时完成（更好的做法是，可以把它放在一个单独的模块中，在编译时使用`use`声明加载这个模块）。然后这个原型在余下的编译阶段中都是可见的。

## 嵌套子例程

如果你（受其他编程语言的影响）习惯于在子例程中使用嵌套的子例程，这些子例程分别有自己的私有变量，要在Perl中做到这一点，则需要多做一些工作。命名子例程不能正确地嵌套，不过匿名子例程可以<sup>注9</sup>。不管怎样，我们可以使用闭包模拟嵌套的词法作用域子例程。下面给出一个例子：

```
sub outer {
    my $x = $_[0] + 35;
    local *inner = sub { return $x * 19 };
    return $x + inner();
}
```

现在由于闭包的临时赋值，`inner`只能从`outer`调用。不过，从`outer`调用时，它会从`outer`的作用域正常地访问词法作用域变量`$x`。

这会产生一个有意思的效果，这样创建的函数对于另一个函数而言是局部函数，正常情况下Perl中并不支持这一点。由于`local`是动态作用域，另外由于函数名对包来说是全局的，所以`outer`调用的所有其他函数也会调用`inner`的临时版本。为避免这一点，你需要多加一个间接层：

```
sub outer {
    my $x = $_[0] + 35;
    my $inner = sub { return $x * 19 };
    return $x + $inner->();
}
```

## 符号引用

如果希望对一个值解引用，但是这个值并不是一个硬引用，会有什么情况发生？这个值会当作一个符号引用（symbolic reference）。也就是说，这个引用将解释为一个表示全局变量名的字符串。

它是这样工作的：

```
$name = "bam";
$$name = 1;          # 设置$bam
$name->[0] = 4;       # 设置@bam的第一个元素
```

---

注9： 更准确地讲，全局命名子例程根本不能嵌套。遗憾的是，这也是我们唯一的命名子例程声明。目前还没有实现词法作用域命名子例程（称为`my subs`），不过等到实现了这种词法作用域命名子例程时，它们应该能正确嵌套了。

```

$name->{X} = "Y";      # 将%bam的X元素设置为Y
@$name = ();          # 清除@bam
keys %$name;          # 得到%bam的键
&$name;               # 调用&bam

```

这个功能很强大，也有些危险，因为有可能原来本想使用一个硬引用，不过不小心使用了一个符号引用。为了避免这种情况，可以写为：

```
use strict "refs";
```

这样一来，外围块中余下的部分只允许使用硬引用。内部代码块可以用以下声明撤销这个限制：

```
no strict "refs";
```

还有一点很重要，要理解下面这两行代码之间的差别：

```

${identifier};        # 等同于$identifier
${"identifier"};      # 也等同于$identifier，不过这是一个符号引用

```

由于第二种形式加了引号，所以会作为一个符号引用，如果启用了`use strict "refs"`，这就会生成一个错误。即使`strict "refs"`未起作用，可能也只是指示一个包变量。不过，第一种形式等同于没有加大括号的形式，如果声明了一个词法作用域变量，它甚至会指向这个变量。下一个例子展示了这一点（下一节将专门讨论）。

只有包变量可以通过符号引用来访问，因为符号引用总是会检查包符号表。由于词法作用域变量不在包符号表中，因此它们对这种机制是不可见的。例如：

```

our $value = "global";
{
    my $value = "private";
    print "Inside, mine is ${value}, ";
    say "but ours is ${"value"}.";
}
say "Outside, ${value} is again ${"value"}.";

```

这会打印以下结果：

```

Inside, mine is private, but ours is global.
Outside, global is again global.

```

## 大括号、中括号和引号

在上一节中，我们指出了`${identifier}`并不作为一个符号引用。你可能想知道它与保留字有什么关系，答案很简单，它们没有关系。尽管`push`是保留字，但下面这两个语句会打印“pop on over”：

```
$push = "pop on ";
```



```
print "${push}over";
```

原因在于，历史上，UNIX shell就是这样使用大括号将变量名与后面的字母数字文本隔离，否则这些文本就会解释为名字的一部分。这是很多人期望的变量内插工作方式，所以在Perl中也是这样做的。不过在Perl中，这个概念得到进一步扩展，还会应用于生成引用时使用的任何大括号，而不论它们是否在引号里面。这说明：

```
print ${push} . "over";
```

或者甚至（因为空格从来都没有影响）：

```
print ${ push } . "over";
```

都会打印“pop on over”，尽管大括号在双引号之外。这个规则同样适用于对散列指定下标所用的任何标识符。所以可以不这样写：

```
$hash{ "aaa" }{ "bbb" }{ "ccc" }
```

可以写为：

```
$hash{ aaa }{ bbb }{ ccc }
```

或

```
$hash{aaa}{bbb}{ccc}
```

而不用担心下标是否是保留字。所以以下代码：

```
$hash{ shift }
```

会解释为\$hash{"shift"}。可以增加任何符号使它不只是一个标识符，从而将它强制解释为一个保留字：

```
$hash{ shift() }  
$hash{ +shift }  
$hash{ shift @_ }
```

## 引用不作为散列键

散列键在内部作为字符串存储<sup>注10</sup>。如果你想把一个引用存储为散列中的键，键值将转换为一个字符串：

```
$x{ \ $a } = $a;  
($key, $value) = each %x;  
print $$key;           # 不正确
```

之前我们提到过，不能将一个字符串反向转换为硬引用。所以如果试图对\$key（其中包含

---

注10：在外部也存储为字符串，如放在DBM文件中时。实际上，DBM文件要求键（和值）必须是字符串。

一个字符串)解引用,并不会返回一个硬解引用,而将返回一个符号解引用,而且由于可能并没有一个名为SCALAR(0x1fc0e)的变量,所以可能得不到你想要的东西。你可能需要这样做:

```
$r = \@a;  
${ $r } = $r;
```

这样至少可以使用散列值(这将是一个硬引用),而不能使用键,键不是硬引用。

尽管不能将引用存储为键,但是如果(如前一个例子)在字符串上下文中使用一个硬引用,它肯定会生成一个唯一的字符串。这是因为,引用的地址会包含在结果字符串中。所以,实际上你可以使用一个引用作为唯一的散列键,只是以后不能对它解引用。

有一种特殊的散列,在这种散列中确实可以使用引用作为键。Perl捆绑提供了一个Tie::RefHash模块,利用它的魔法(magic)<sup>注11</sup>,刚才我们说过不能做的事情也将变得可行:

```
use Tie::RefHash;  
tie my %h, "Tie::RefHash";  
%h = (  
    ["this", "here"] => "at home",  
    ["that", "there"] => "elsewhere",  
);  
while ( my($keyref, $value) = each %h ) {  
    say "@$keyref is $value";  
}
```

实际上,通过为内置类型绑定不同的实现,可以使标量、散列和数组拥有很多我们从前说无法做到的行为。别想骗过我们!

关于绑定的更多内容,请参见第14章。

## 垃圾回收、循环引用和弱引用

高级语言通常都允许程序员用完某个内存时不用担心内存的释放。这种自动回收过程称为垃圾回收(garbage collection)。对于大多数情况,Perl会使用一种快速而简单的基于引用的垃圾回收器。

退出一个代码块时,它的局部作用域变量会正常释放,不过有可能隐藏你的垃圾,使Perl的垃圾回收器找不到它们。这有可能导致一个严重的问题:引用数为0的不可达的内存通常无法释放。因此,循环引用是个很糟糕的想法:

```
{ # 使$a和$b相互指向对方
```

---

注11: 没错,这确实是一个专业术语,如果查看Perl源代码发布版本中的mg.c文件,你会注意到这一点。

```

    my ($a, $b);
    $a = \$b;
    $b = $a;
}

```

或者写为：

```

{
    # 使$a指向自己
    my $a;
    $a = $a;
}

```

尽管\$a应当在块结束时释放，但实际上并没有释放。构造递归的数据结构时，如果希望在程序（或线程）退出之前回收内存，你就必须自行断开（或弱化，见下面的介绍）这种自引用（退出时，会通过一个开销很大但完整的标记-清除垃圾回收机制为你自动回收内存）。如果数据结构是一个对象，可以使用DESTROY方法来自动断开引用，参见第12章中的“使用DESTROY方法完成垃圾回收”一节。

缓存也可能出现类似的情况，缓存（cache）是一种数据存储仓库，设计用来以更快的速度获取数据。有一些缓存外的引用会指向缓存内的数据。如果所有这些引用都已经删除，但是缓存数据及其内部引用仍保留，这就会出现这个问题。尽管我们希望缓存数据一旦不再需要它们就消失，但由于存在引用，这会阻止Perl回收指示对象。与循环引用类似，这里需要一个不影响引用计数的引用，从而不会延迟垃圾回收。

下面再看另一个例子，这一次给出一个显式的循环双向链表：

```

$ring = {
    VALUE => undef,
    NEXT => undef,
    PREV => undef,
};
$ring->{NEXT} = $ring;
$ring->{PREV} = $ring;

```

底层散列的底层引用数为3，通过将\$ring置为undef，或者让它退出作用域，这样做只会将这个引用计数减1，将导致Perl无法恢复整个散列内存。

为了解决这种情况，Perl引入了弱引用（weak references）的概念。弱引用就像其他常规的引用一样（这是指“硬”引用，而不是“符号”引用），只是另外有两个重要的特性：它不再影响其指示对象的引用计数，另外当弱引用的指示对象被垃圾回收时，弱引用本身会变为未定义。这些特性使弱引用非常适用于那些包含有自身内部引用的数据结构。这样一来，这些内部引用不会计入结构的引用数，不过仍会统计外部引用数。

尽管Perl从v5.6就开始支持弱引用，但是并没有提供标准的weaken函数从Perl自身访问这些弱引用，直到v5.8.1版本才首次将weaken函数作为标准在Scalar::Util模块中引入。这个模块还提供了一个is\_weak函数，它会报告其引用参数是否是一个弱引用。



对于前面的ring例子，可以使用弱引用实现，如下所示：

```
use Scalar::Util qw(weaken);

$ring = {
    VALUE => undef,
    NEXT => undef,
    PREV => undef,
};
$ring->{NEXT} = $ring;
$ring->{PREV} = $ring;
weaken($ring->{NEXT});
weaken($ring->{PREV});
```

就ref操作符而言，弱引用的工作类似于正常的（硬）引用：它会报告指示对象的类型。不过，一个弱引用的指示对象被垃圾回收时，包含这个弱引用的变量会突然变成未定义，因为它不再指示一个真正存在的东西。

复制一个弱引用只会创建一个常规引用。如果需要另外一个弱引用，必须在创建引用之后完成弱化。

如果需要一个使用弱引用的更大的例子，参见《Perl Cookbook》中技巧11.15 “Coping with Circular Data Structures using Weak References”（使用弱引用复制循环数据结构）。

# 数据结构

Perl免费提供了很多数据结构，在其他编程语言中，这些数据结构往往需要你自己来构建。计算机专家们都学过堆栈和队列，而在Perl中堆栈和队列都只是数组。使用push和pop（或unshift和shift）压入和弹出一个数组时，它就是一个堆栈；使用push和shift（或unshift和pop）处理数组时，这就是一个队列。另外很多树结构的建立只是为了给概念上扁平的查找表提供快速而动态的访问。实际上，散列作为Perl内置的结构，完全可以为概念上扁平的查找表提供快速动态的访问，而不需要那些头脑发热的人所津津乐道的复杂递归数据结构。

不过，有时你可能确实需要一些嵌套数据结构，因为这些数据结构可以最为自然地为你要解决的问题建模。所以Perl允许结合和嵌套数组和散列来创建任意复杂的数据结构。如果使用得当，可以用它们创建链表、二叉树、堆、B-树、集合、图以及你能想到的任何结构。可以参阅《Mastering Algorithms with Perl》、《Perl Cookbook》、perldsc中的“Data Structure Cookbook”或者CPAN（这是所有这些模块的中央存储库）。不过，实际上你可能只需要简单地结合数组和散列，这也是本章将要讨论的内容。

## 数组的数组

有很多不同类型的嵌套数据结构。最简单的就是数组的数组（array of arrays），也称为二维数组或矩阵（显然可以进一步推广：数组的数组的数组就是一个三维数组，依此类推，可以得到更高维的数组）。这很容易理解，这里使用的原则几乎都同样适用于后面各节将要讨论的更有趣的数据结构。

## 创建和访问二维数组

可以如下建立一个二维数组：

```
# 将一个数组引用列表赋至一个数组
@AoA = (
    [ "fred", "barney" ],
    [ "george", "jane", "elroy" ],
    [ "homer", "marge", "bart" ],
);

print $AoA[2][1]; # 打印"marge"
```

整个列表用小括号包围，而不是中括号，因为你要为@AoA赋一个列表而不是引用。如果想要为变量赋一个数组引用，则使用中括号：

```
# 创建一个数组的引用，这个数组由其他数组引用组成
$ref_to_AoA = [
    [ "fred", "barney", "pebbles", "bamm bamm", "dino", ],
    [ "homer", "bart", "marge", "maggie", ],
    [ "george", "jane", "elroy", "judy", ],
];

print $ref_to_AoA->[2][3]; # 打印"judy"
```

要记住，在每对相邻的大括号或中括号之间有一个隐含的->。因此，下面这两行代码：

```
$AoA[2][3]
$ref_to_AoA->[2][3]
```

等价于下面的两行代码：

```
$AoA[2]->[3]
$ref_to_AoA->[2]->[3]
```

不过，第一对中括号前面没有隐含的->，正是因为这个原因，\$ref\_to\_AoA的解引用需要有开头的->。另外要记住，可以用一个负索引从数组末尾倒数，所以：

```
$AoA[0][-2]
```

是第一行的倒数第二个元素。

## 扩展你自己的数组

这种大列表赋值很适合创建固定的数据结构，不过，如果你想动态计算各个元素该怎么做呢？或者如果你想一部分一部分地构建这个结构，又该怎么做呢？



下面从一个文件读入一个数据结构。这里假设这是一个纯文本文件，其中每一行文本分别是结构中的一行，每行文本包含由空白符分隔的元素。方法如下<sup>注1</sup>：

```
while (<>) {
    @tmp = split;          # 将元素分解到一个数组中
    push @AoA, [ @tmp ];   # 向@AoA增加一个匿名数组引用
}
```

当然，你不需要对这个临时数组命名，所以也可以写为：

```
while (<>) {
    push @AoA, [ split ];
}
```

如果希望增加一个数组的数组的引用，可以写为：

```
while (<>) {
    push @$ref_to_AoA, [ split ];
}
```

这两个例子都是向数组的数组增加新行。如果要增加新的列呢？如果只是处理二维数组，通常最容易的方法是使用简单赋值<sup>注2</sup>：

```
for $x (0 .. 9) {
    for $y (0 .. 9) {
        $AoA[$x][$y] = func($x, $y); # ...设置这个单元
    }
}

for $x ( 0..9 ) {
    $ref_to_AoA->[$x][3] = func2($x); # ...设置第4列
}
```

采用什么顺序为元素赋值并不重要，@AoA的下标元素是否存在也不重要。Perl会很高兴地为你创建这些元素，还会根据需要将中间的一些元素设置为未定义值。在前面的代码中，如果需要，Perl甚至还会为你在\$ref\_to\_AoA中创建初始引用。如果你只想为一行追加新的列，则需要做一些更有意思的工作：

```
# 为已有行追加新列
push @{ $AoA[0] }, "wilma", "betty";
```

你可能想知道能不能跳过这个解引用，只写为：

```
push $AoA[0], "wilma", "betty"; # 编译错误 < v5.14
```

---

注1： 与其他章节中一样，在这里我们（为简洁起见）省略了my声明，不过正常情况下还是应当加上my声明。在这个例子中，通常应当写为my @tmp = split。

注2： 与前面的临时赋值一样，这里有所简化：这一章中的循环在实际代码中应当写为for my \$x。

我们自己也同样很困惑。大多数情况下，这甚至不能编译，因为`push`的参数必须是一个真正的数组，而不能只是一个数组引用。因此，它的第一个参数必须以一个`@`字符开头，不过对`@`后面的部分并没有太多要求。

在v5.14中，有时可以通过调用某些内置函数来省略显式的解引用。这些函数包括：对于数组有`pop`、`push`、`shift`、`unshift`和`splice`，对于散列有`keys`、`values`和`each`。这些函数不再要求第一个参数以`@`或`%`开头。如果传入一个适当类型集合的合法引用，这些函数会根据需要将它解引用。与显式解引用不同，这种隐式解引用不会触发自动生成（`autovivification`）。如果传入一个非法的引用，会产生一个运行时异常。如果在较老的版本上运行你的漂亮的新代码，会让这些“老态龙钟”的编译器接受不了，所以你要在文件最前面放上一个`use VERSION pragma`，通知用户这是新版本的代码：

```
use 5.014;    # 旧瓶无新酒
use v5.14;    # 旧布没有新补丁
```

## 访问和打印

下面来打印这个数据结构。如果你只想要一个元素，下面的代码就足够了：

```
print $AoA[3][2];
```

如果你想打印整个结构，不能简单地写为：

```
print @AoA;          # 不正确
```

这样是不对的，因为你只会看到字符串化的引用，而不是结构中的实际数据。Perl绝对不会为你自动解引用。实际上，你必须使用一、两个循环来访问数据。下面的代码会打印整个结构，它会循环处理`@AoA`的元素，并在`print`语句中对各个元素分别解引用：

```
for $row ( @AoA ) {
    say "$row";
}
```

如果想跟踪下标，可以这样做：

```
for $i ( 0 .. $#AoA ) {
    say "row $i is: @{$AoA[$i]}";
}
```

或者甚至可以这样做（注意内循环）：

```
for $i ( 0 .. $#AoA ) {
    for $j ( 0 .. ${#AoA[$i]} ) {
        say "element $i $j is $AoA[$i][$j]";
    }
}
```

可以看到，开始有点复杂了，所以有时更容易的做法是在循环中使用一个临时变量：

```

for $i ( 0 .. $#AoA ) {
    $row = $AoA[$i];
    for $j ( 0 .. ${$row} ) {
        say "element $i $j is $row->[$j]";
    }
}

```

你可能厌倦了这样编写定制代码来打印你的数据结构，那么可以看看标准Dumpvalue或Data::Dumper模块。前一个模块正是Perl调试器使用的模块，后者会生成可解析的Perl代码。例如：

```

use v5.14;          # 使用+原型 (v5.14新增)

sub show(+) {
    require Dumpvalue;
    state $prettily = new Dumpvalue::
        tick          => q(""),
        compactDump => 1,  #如果你想要更大的转储数据,
        veryCompact => 1,  # 注释掉这两行代码
    ;
    dumpValue $prettily @_;
}

# 将一个数组引用列表赋至一个数组
my @AoA = (
    [ "fred", "barney" ],
    [ "george", "jane", "elroy" ],
    [ "homer", "marge", "bart" ],
);
push $AoA[0], "wilma", "betty";
show @AoA;

```

这会打印以下结果：

```

0 0..3 "fred" "barney" "wilma" "betty"
1 0..2 "george" "jane" "elroy"
2 0..2 "homer" "marge" "bart"

```

不过，如果正像我们所说的，如果你想要更大的转储数据，将那两行代码注释掉，则会显示数组内容，如下所示：

```

0 ARRAY(0x8031d0)
  0 "fred"
  1 "barney"
  2 "wilma"
  3 "betty"
1 ARRAY(0x803d40)
  0 "george"
  1 "jane"
  2 "elroy"
2 ARRAY(0x803e10)
  0 "homer"
  1 "marge"

```



```
2 "bart"
```

要显示数据转储，我们想使用CPAN模块Data::Dump。用法如下：

```
use v5.14;                # 对于标量的push操作
use Data::Dump qw(dump);  # CPAN模块

my @AoA = (
    [ "fred", "barney" ],
    [ "george", "jane", "elroy" ],
    [ "homer", "marge", "bart" ],
);
push $AoA[0], "wilma", "betty";
dump \@AoA;
```

这会生成以下输出：

```
[
  ["fred", "barney", "wilma", "betty"],
  ["george", "jane", "elroy"],
  ["homer", "marge", "bart"],
]
```

## 片段

如果你想访问一个多维数组中的一个片段（行的一部分），必须做一些有意思的下标处理。利用指针箭头可以访问单个元素，但是对于片段却没有这种便利方法。当然可以使用一个循环逐个地取出片段中的各个元素：

```
@part = ();
for ($y = 7; $y < 13; $y++) {
    push @part, $AoA[4][$y];
}
```

这个循环可以替换为一个数组片段：

```
@part = @{ $AoA[4] } [ 7..12 ];
```

如果你想要一个二维片段（two-dimensional slice），比如说，\$x从4到8，\$y从7到12，下面给出一种做法：

```
@newAoA = ();
for ($startx = $x = 4; $x <= 8; $x++) {
    for ($starty = $y = 7; $y <= 12; $y++) {
        $newAoA[$x - $startx][$y - $starty] = $AoA[$x][$y];
    }
}
```

在这个例子中，会逐个地为目标二维数组@newAoA赋各个值，这些值取自一个二维子数组@AoA。另一种做法是创建匿名数组，这些匿名数组分别包含@AoA子数组中所要的一个片段，然后把这些匿名数组的引用放入@newAoA。这样一来，可以将引用写入@newAoA（也就

是说，只设置一次下标），而不是把子数组的值写入@newAoA（这需要设置两重下标）。这个方法可以消除最内层循环：

```
for ($x = 4; $x <= 8; $x++) {  
    push @newAoA, [ @{ $AoA[$x] } [ 7..12 ] ];  
}
```

当然，如果你需要经常这么做，可能应当写一个子例程，比如名为extract\_rectangle的一个子例程。如果你需要很频繁地对大集合或多维数据做这种处理，可以使用CPAN上提供的一个PDL（Perl Data Language，Perl数据语言）模块。

## 常见错误

前面已经提到，Perl数组和散列都是一维的。在Perl中，实际上甚至“多维”数组也是一维的，不过这一维的值是其他数组的引用，它会把多个元素合并到一个数组中。如果打印这些值而没有对它们解引用，就会得到字符串化的引用，而不是你想要的数据。例如，下面这两行代码：

```
@AoA = ( [2, 3], [4, 5, 7], [0] );  
print "@AoA";
```

会得到类似这样的结果：

```
ARRAY(0x83c38) ARRAY(0x8b194) ARRAY(0x8b1d0)
```

另一方面，下面这行代码会显示7：

```
print $AoA[1][2];
```

构造一个数组的数组时，记住要为子数组生成新引用。否则，你只会创建一个数组，其中包含子数组的元素个数，如下：

```
for $i (1..10) {  
    @array = somefunc($i);  
    $AoA[$i] = @array; # 不正确!  
}
```

这里会在标量上下文中访问@array，因此会得到其元素个数，然后再把得到的元素个数赋至\$AoA[\$i]。稍后会介绍赋引用的正确做法。

犯了前面这个错误之后，人们开始意识到需要赋一个引用，所以又会很自然地犯另一个错误：反复地使用同一个内存位置的引用：

```
for $i (1..10) {  
    @array = somefunc($i);  
    $AoA[$i] = \@array;      # 还是不正确!  
}
```

for循环中第二行生成的每一个引用都是一样的，确切地讲，它们都是数组@array的引用。没错，每一次循环时这个数组都会改变，不过一切结束时，\$AoA将包含指向同一个数组的10个引用，这个数组包含最后赋给它的一组值。print@{\$AoA[1]}与print@{\$AoA[2]}显示的值是一样的。

下面来看一种更可取的方法：

```
for $i (1..10) {  
    @array = somefunc($i);  
    $AoA[$i] = [ @array ]; # 正确!  
}
```

@array两边的中括号会创建一个新的匿名数组，并把@array的元素复制到这个匿名数组中，然后存储这个新数组的引用。

以下代码会得到类似的结果，不过这个代码可读性要差一些：

```
for $i (1..10) {  
    @array = somefunc($i);  
    @{$AoA[$i]} = @array;  
}
```

由于\$AoA[\$i]必须是一个新引用，所以这个引用会自然生成。然后前面的@解引用这个新引用，其结果是（在列表上下文中）@array的值将赋至\$AoA[\$i]引用的数组。出于简洁性考虑，你可能想避免这种做法。

不过，在某种情况下可能确实需要用到这种做法。假设@AoA已经是一个包含数组引用的数组。也就是说，你已经完成了以下赋值：

```
$AoA[3] = \@original_array;
```

现在假设你想改变@original\_array（也就是说，你希望修改\$AoA的第4行），让它指向@array的元素。下面的代码是可用的：

```
@{$AoA[3]} = @array;
```

在这种情况下，引用本身并不改变，但是所引用数组中的元素确实会改变。这会覆盖@original\_array的值。

最后需要说明，下面这个代码尽管看上去有些危险，但它确实能很好地工作：

```
for $i (1..10) {  
    my @array = somefunc($i);  
    $AoA[$i] = \@array;  
}
```

这是因为，每次循环时都会创建全新的词法作用域my @array变量。所以，尽管看上去每次都在存储同一个变量引用，但实际上并非如此。这是一个很微小的差别，不过利用这个



技术可以得到更高效的代码，但有可能误导水平稍差的程序员（这种代码之所以更高效，是因为最后的赋值中没有复制值）。另一方面，如果必须复制值（循环中的第一个赋值就要这样做），可以使用中括号隐含创建的副本，而避免使用临时变量：

```
for $i (1..10) {  
    $AoA[$i] = [ somefunc($i) ];  
}
```

总结一下：

```
$AoA[$i] = [ @array ]; # 最安全，有时也最快  
$AoA[$i] = \@array;    # 快速，但有风险，取决于数组的my声明  
@{ $AoA[$i] } = @array; # 有点复杂
```

一旦掌握了数组的数组，你可能还想处理更复杂的数据结构。如果想在Perl中查找C结构或Pascal记录，你会发现Perl中并没有特殊的保留字来建立这些结构。实际上，你会得到一个更为灵活的系统。如果你对记录结构的认识还比较死板，不太灵活，或者如果你想为用户提供更固化、更严格的结构，那么你可以使用面向对象特性，这个内容将在第12章详细介绍。

Perl只有两种组织数据的方法：作为有序列表存储在数组中，并按位置访问；或者作为无序的键/值对存储在散列中，按名字访问。在Perl中，表示一个记录的最佳方法是利用散列引用，不过如何组织这些记录则没有固定的方法。你可能想维护这些记录的一个有序列表，可以按编号查找，在这种情况下需要使用一个散列引用数组来存储记录。或者，你可能希望按名字查找记录，这种情况下则需要维护一个散列引用的散列。

下面几节中，你会看到一些代码示例将详细介绍如何从头开始生成、从其他来源生成、访问和显示多种不同的数据结构。首先我们将展示数组和散列的3种直接组合方式，然后介绍函数散列和不太规则的数据结构。最后将说明如何保存这些数据结构。介绍这些例子之前，我们假设你已经非常熟悉这一章前面解释的内容。

## 数组的散列

如果你想根据一个特定的字符串查找各个数组，而不只是利用一个索引编号来查找，就应当使用数组的散列。在我们的电视剧角色例子中，并不是按第0个电视剧、第1个电视剧等查找人名列表，我们会适当地建立这个结构，以便在给定剧名时能够查找相应的演员列表。由于外层数据结构是一个散列，所以无法对内容排序，不过可以使用sort函数指定一个特殊的输出顺序。

## 创建数组的散列

可以如下创建一个匿名数组散列：

```
# 如果键是标识符，我们通常会省略引号
%HoA = (
    flintstones    => [ "fred", "barney" ],
    jetsons        => [ "george", "jane", "elroy" ],
    simpsons       => [ "homer", "marge", "bart" ],
);
```

要向这个散列增加另一个数组，可以写为：

```
$HoA{teletubbies} = [ "tinky winky", "dipsy", "laa-laa", "po" ];
```

## 生成数组的散列

可以利用下面的技术填充数组的散列。要读取有以下格式的文件：

```
flintstones: fred barney wilma dino
jetsons:      george jane elroy
simpsons:     homer marge bart
```

可以使用以下两个循环中的任意一个：

```
while ( <> ) {
    next unless s/^(.*?):\s*//;
    $HoA{$1} = [ split ];
}

while ( $line = <> ) {
    ($who, $rest) = split /\s*/, $line, 2;
    @fields = split " ", $rest;
    $HoA{$who} = [ @fields ];
}
```

如果有一个返回数组的子例程`get_family`，可以利用这个子例程采用以下任意一个循环填充%HoA：

```
for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoA{$group} = [ get_family($group) ];
}

for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoA{$group} = [ @members ];
}
```

还可以向一个现有的数组追加新成员，如下所示：

```
push @{ $HoA{flintstones} }, "wilma", "pebbles";
```

## 访问和打印数组的散列

可以如下设置一个特定数组的第一个元素：

```
$HoA{flintstones}[0] = "Fred";
```

要将第二个Simpson首字母大写，需要对适当的数组元素应用一个替换：

```
$HoA{simpsons}[1] =~ s/(\w)/\u$1/;
```

可以循环处理散列的键来打印所有家庭：

```
for $family ( keys %HoA ) {  
    say "$family: @{ $HoA{$family} }";  
}
```

稍做一些努力，还可以增加数组索引：

```
for $family ( keys %HoA ) {  
    print "$family: ";  
    for $i ( 0 .. ${ $HoA{$family} } ) {  
        print " $i = $HoA{$family}[$i]";  
    }  
    print "\n";  
}
```

或者可以按数组中包含的元素个数对这些数组排序：

```
for $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {  
    say "$family: @{ $HoA{$family} }"  
}
```

甚至可以按元素个数对数组排序，然后按ASCII字母顺序（准确地讲，是utf8顺序）对元素排序：

```
# 打印按成员个数和名字排序的整个数组  
for $family ( sort { @{$HoA{$b}} <=> @{$HoA{$a}} } keys %HoA ) {  
    say "$family: ", join(" ", => sort @{$HoA{$family}});  
}
```

如果家庭成员名字中有非ASCII的Unicode字符，甚至有某种标点符号，按码点顺序排序并不会生成一个字母表顺序。应当这样做：

```
use Unicode::Collate;  
my $sorter = Unicode::Collate->new(); # 正常的字母表顺序排序  
say "$family: ",  
    join " ", " => $sorter->sort( @{$HoA{$family} } );
```

## 散列的数组

如果有一组记录，你希望顺序访问这些记录，而且每个记录本身包含有键/值对，这种情况下，散列的数组会很有用。与本章介绍的其他结构相比，散列的数组使用不算太多。



## 创建散列的数组

可以如下创建一个匿名散列的数组：

```
@AoH = (
    {
        husband => "barney",
        wife => "betty",
        son => "bamm bamm",
    },
    {
        husband => "george",
        wife => "jane",
        son => "elroy",
    },
    {
        husband => "homer",
        wife => "marge",
        son => "bart",
    },
);
```

要为这个数组增加其他散列，可以写为：

```
push @AoH, { husband => "fred", wife => "wilma", daughter => "pebbles" };
```

## 生成散列的数组

可以利用下面的技术填充一个散列的数组。要读取有以下格式的文件：

```
husband=fred friend=barney
```

可以使用以下两个循环中的任意一个：

```
while ( <> ) {
    $rec = {};
    for $field ( split ) {
        ($key, $value) = split /=/, $field;
        $rec->{$key} = $value;
    }
    push @AoH, $rec;
}

while ( <> ) {
    push @AoH, { split /\s=/+ };
}
```

如果有一个返回键/值对的子例程`get_next_pair`，可以利用这个子例程采用以下任意一个循环填充`@AoH`：

```
while ( @fields = get_next_pair() ) {
    push @AoH, { @fields };
}
```

```

}

while (<>) {
    push @AoH, { get_next_pair($_) };
}

```

可以向一个现有的散列追加新成员，如下所示：

```

$AoH[0]{pet} = "dino";
$AoH[2]{pet} = "santa's little helper";

```

## 访问和打印散列的数组

可以如下设置一个特定散列的一个键/值对：

```

$AoH[0]{husband} = "fred";

```

要将第二个数组的husband首字母大写，需要应用一个替换：

```

$AoH[1]{husband} =~ s/(\w)/\u$1/;

```

可以如下打印所有数据：

```

for $href ( @AoH ) {
    print "{ ";
    for $role ( keys %$href ) {
        print "$role=$href->{$role} ";
    }
    print "}\n";
}

```

也可以利用索引来打印：

```

for $i ( 0 .. $#AoH ) {
    print "$i is { ";
    for $role ( keys %{ $AoH[$i] } ) {
        print "$role=$AoH[$i]{$role} ";
    }
    print "}\n";
}

```

## 散列的散列

多维散列是Perl中最灵活的嵌套结构。这就像构建一个记录，而这个记录本身又包含其他记录。在每个层次上，要用字符串（如果必要则需要加引号）来索引这个散列。不过要记住，散列中的键/值对并没有任何特定的顺序，可以使用sort函数按你喜欢的顺序获取这些键/值对。

## 创建散列的散列

可以如下创建匿名散列的散列：

```
%HoH = (  
  flintstones => {  
    husband => "fred",  
    pal => "barney",  
  },  
  jetsons => {  
    husband => "george",  
    wife => "jane",  
    "his boy" => "elroy", # 需要为键加引号  
  },  
  simpsons => {  
    husband => "homer",  
    wife => "marge",  
    kid => "bart",  
  },  
);
```

要为%HoH增加另一个匿名散列，可以写为：

```
$HoH{ mash } = {  
  captain => "pierce",  
  major => "burns",  
  corporal => "radar",  
};
```

## 生成散列的散列

可以使用下面的技术填充一个散列的散列。要读取有以下格式的文件：

```
flintstones: husband=fred pal=barney wife=wilma pet=dino
```

可以使用以下两个循环中的任意一个：

```
while ( <> ) {  
  next unless s/^(.*?):\s*//;  
  $who = $1;  
  for $field ( split ) {  
    ($key, $value) = split /=/, $field;  
    $HoH{$who}{$key} = $value;  
  }  
}  
  
while ( <> ) {  
  next unless s/^(.*?):\s*//;  
  $who = $1;  
  $rec = {};  
  $HoH{$who} = $rec;  
  for $field ( split ) {  
    ($key, $value) = split /=/, $field;
```



```

        $rec->{$key} = $value;
    }
}

```

如果有一个返回键/值对列表的子例程`get_family`，可以利用这个子例程采用以下3个代码段中的任意一个循环填充`%HoH`：

```

for $group ( "simpsons", "jetsons", "flintstones" ) {
    $HoH{$group} = { get_family($group) };
}

for $group ( "simpsons", "jetsons", "flintstones" ) {
    @members = get_family($group);
    $HoH{$group} = { @members };
}

sub hash_families {
    my @ret;
    for $group ( @_ ) {
        push @ret, $group, { get_family($group) };
    }
    return @ret;
}
%HoH = hash_families( "simpsons", "jetsons", "flintstones" );

```

可以如下为一个现有的散列追加新成员：

```

%new_folks = (
    wife => "wilma",
    pet => "dino";
);
for $what (keys %new_folks) {
    $HoH{flintstones}{$what} = $new_folks{$what};
}

```

## 访问和打印散列的散列

可以如下设置一个特定散列中的一个键/值对：

```
$HoH{flintstones}{wife} = "wilma";
```

要将一个特定的键/值对首字母大写，可以对元素应用一个替换：

```
$HoH{jetsons}{his boy} =~ s/(\w)/\u$1/;
```

先循环处理外散列的键，然后循环处理内散列的键，这样可以打印所有家庭：

```

for $family ( keys %HoH ) {
    print "$family: ";
    for $role ( keys %{ $HoH{$family} } ) {
        print "$role=$HoH{$family}{$role} ";
    }
    print "\n";
}

```

```
}
```

在非常大的散列中，使用each（这样能避免排序）同时获取键和值可能会稍快一些：

```
while ( ($family, $roles) = each %HoH ) {  
    print "$family: ";  
    while ( ($role, $person) = each %$roles ) {  
        print "$role=$person ";  
    }  
    print "\n";  
}
```

遗憾的是，大散列才真正需要排序，否则在打印输出里永远也找不到你真正想找的东西。可以先按家庭再按角色进行排序，如下所示：

```
for $family ( sort keys %HoH ) {  
    print "$family: ";  
    for $role ( sort keys %{ $HoH{$family} } ) {  
        print "$role=$HoH{$family}{$role} ";  
    }  
    print "\n";  
}
```

要按成员个数对家庭排序（而不是ASCII字母表顺序[或utf8顺序]），可以在标量上下文中使用keys：

```
for $family ( sort { keys %{ $HoH{$a} } <=> keys %{ $HoH{$b} } } keys %HoH ) {  
    print "$family: ";  
    for $role ( sort keys %{ $HoH{$family} } ) {  
        print "$role=$HoH{$family}{$role} ";  
    }  
    print "\n";  
}
```

要以某个固定的顺序对家庭成员排序，可以分别为每个成员指定等级（rank）：

```
$i = 0;  
for ( qw(husband wife son daughter pal pet) ) { $rank{$_} = ++$i }  
  
for $family ( sort { keys %{ $HoH{$a} } <=> keys %{ $HoH{$b} } } keys %HoH ) {  
    print "$family: ";  
    for $role ( sort { $rank{$a} <=> $rank{$b} } keys %{ $HoH{$family} } ) {  
        print "$role=$HoH{$family}{$role} ";  
    }  
    print "\n";  
}
```

## 函数的散列

用Perl编写一个复杂的应用或网络服务时，你可能希望为用户提供大量命令，供他们使用。这样一个程序可能会有类似下面的代码，来检查用户的选择并采取适当的动作：

```

if ($cmd =~ /^exit$/i) { exit }
elsif ($cmd =~ /^help$/i) { show_help() }
elsif ($cmd =~ /^watch$/i) { $watch = 1 }
elsif ($cmd =~ /^mail$/i) { mail_msg($msg) }
elsif ($cmd =~ /^edit$/i) { $edited++; editmsg($msg); }
elsif ($cmd =~ /^delete$/i) { confirm_kill() }
else {
    warn "Unknown command: '$cmd'; Try 'help' next time\n";
}

```

也可以将函数引用存储在你的数据结构中，就像存储数组或散列的引用一样：

```

%HoF = (                                # 创建一个函数的散列
    exit => sub { exit },
    help => \&show_help,
    watch => sub { $watch = 1 },
    mail => sub { mail_msg($msg) },
    edit => sub { $edited++; editmsg($msg); },
    delete => \&confirm_kill,
);

if ($HoF{lc $cmd}) { $HoF{lc $cmd}->() } # 调用函数
else { warn "Unknown command: '$cmd'; Try 'help' next time\n" }

```

在倒数第二行中，我们检查指定的命令名（小写）是否在“分派表”%HoF中。如果存在，则将散列值解引用为一个函数，来调用适当的命令，并为这个函数传入一个空参数表。也可以把它解引用为&{ \$HoF{lc \$cmd} }()，或者在Perl的v5.6版本中，可以简单地解引用为\$HoF{lc \$cmd}()。

## 更复杂的记录

到目前为止，这一章我们看到的都是很简单的两层同构数据结构：每个元素与该层次上所有其他元素一样都包含相同类型的指示对象。并不一定非得如此。任何元素都可以包含任意类型的标量，这说明可以是一个字符串、一个数字或者是某个引用。这个引用可以是一个数组或散列引用，也可以是一个命名或匿名函数的引用，还可以是对象的引用。只有一点是不允许的：不能把多个指示对象塞在一个标量里。如果你发现必须这么做，这说明你可能需要一个数组或散列引用，要把多个值“压缩”在一个数组或散列中。

下一节中会给出一些代码示例，通过专门设计，这些例子展示了一个记录中可以存储多种不同类型的数据，这里我们使用一个散列引用来实现这样一个记录。散列的键是大写的字符串，这是一个约定，使用散列作为特定记录类型时经常会采用这个约定（有时也可能不遵循这个约定，不过这种情况很少）。

## 更复杂记录的组合、访问和打印

下面这个记录包含6个不同的字段：



```
$rec = {
    TEXT => $string,
    SEQUENCE => [ @old_values ],
    LOOKUP => { %some_table },
    THATCODE => \&some_function,
    THISCODE => sub { $_[0] ** $_[1] },
    HANDLE => \*STDOUT,
};
```

*TEXT*字段是一个简单的字符串，所以可以直接打印：

```
print $rec->{TEXT};
```

*SEQUENCE*和*LOOKUP*是常规的数组和散列引用：

```
print $rec->{SEQUENCE}[0];
$last = pop @{$rec->{SEQUENCE}};
print $rec->{LOOKUP}{"key"};
($first_k, $first_v) = each %{$rec->{LOOKUP}};
```

*THATCODE*是一个命名子例程，*THISCODE*是一个匿名子例程，不过它们的调用都是一样的：

```
$that_answer = $rec->{THATCODE}->($arg1, $arg2);
$this_answer = $rec->{THISCODE}->($arg1, $arg2);
```

通过多加了一对大括号，可以把*\$rec->{HANDLE}*看作一个间接对象：

```
print { $rec->{HANDLE} } "a string\n";
```

如果使用*IO::Handle*模块，甚至可以把这个句柄看作是一个常规对象：

```
use IO::Handle;
$rec->{HANDLE}->autoflush(1);
$rec->{HANDLE}->print("a string\n");
```

## 更加复杂的记录的组合、访问和打印

很自然地，数据结构的字段本身也可以是任意复杂的数据结构：

```
%TV = (
    flintstones => {
        series => "flintstones",
        nights => [ "monday", "thursday", "friday" ],
        members => [
            { name => "fred", role => "husband", age => 36, },
            { name => "wilma", role => "wife", age => 31, },
            { name => "pebbles", role => "kid", age => 4, },
        ],
    },
    jetsons => {
        series => "jetsons",
        nights => [ "wednesday", "saturday" ],
    },
);
```

```

        members => [
            { name => "george", role => "husband", age => 41, },
            { name => "jane", role => "wife", age => 39, },
            { name => "elroy", role => "kid", age => 9, },
        ],
    },
    simpsons => {
        series => "simpsons",
        nights => [ "monday" ],
        members => [
            { name => "homer", role => "husband", age => 34, },
            { name => "marge", role => "wife", age => 37, },
            { name => "bart", role => "kid", age => 11, },
        ],
    },
};

```

## 复杂记录散列的生成

由于解析复杂数据结构是Perl的长项，你可以把数据声明与常规的Perl代码分开，放在一个单独的文件中，然后用`do`或`require`内置函数加载这些数据。另外一种常用的方法是使用一个CPAN模块（如`XML::Parser`）来加载用另外某种语言（如XML）表示的任意数据结构。

可以分片地构建数据结构：

```

$rec = {};
$rec->{series} = "flintstones";
$rec->{nights} = [ find_days() ];

```

或者从一个文件读取（在这里，假设文件采用`field=value`语法格式）：

```

@members = ();
while (<>) {
    %fields = split /\s=/+;
    push @members, { %fields };
}
$rec->{members} = [ @members ];

```

再把它们组合到更大的数据结构中（以某个子字段为键）：

```

$TV{ $rec->{series} } = $rec;

```

可以使用额外的指针字段来避免重复数据。例如，你可能希望一个人的记录中包含一个`"kids"`字段，这可能是指向一个数组的引用，该数组本身又包含多个引用，这些引用分别指向各个孩子自己的记录。通过让数据结构的某些部分指示另外一些部分，可以避免由于某处更新了数据而另外一处未更新数据所导致的数据不一致：

```

for $family (keys %TV) {

```

```

my $rec = $TV{$family}; # 临时指针
@kids = ();
for $person ( @{$rec->{members}} ) {
    if ($person->{role} =~ /kid|son|daughter/) {
        push @kids, $person;
    }
}
# $rec 和 $TV{$family} 指向相同的数据!
$rec->{kids} = [ @kids ];
}

```

`$rec->{kids} = [ @kids ]`赋值会复制数组内容。不过它们只是数据的引用，而数据并没有复制。这说明，如果增加Bart的年龄，如下所示：

```
$TV{simpsons}{kids}[0]{age}++; # 增加到12
```

你会看到以下结果，因为`$TV{simpsons}{kids}[0]`和`$TV{simpsons}{members}[2]`都指向同一个底层匿名散列表：

```
print $TV{simpsons}{members}[2]{age}; # 也打印12
```

现在打印整个%TV结构：

```

for $family ( keys %TV ) {
    print "the $family";
    print " is on ", join (" and ", @{$TV{$family}{nights}}), "\n";
    print "its members are:\n";
    for $who ( @{$TV{$family}{members}} ) {
        print " $who->{name} ($who->{role}), age $who->{age}\n";
    }
    print "children: ";
    print join (" ", map { $_->{name} } @{$TV{$family}{kids}} );
    print "\n\n";
}

```

## 保存数据结构

如果你想保存你的数据结构，以便其他程序以后使用，这有很多种方法。最容易的方法是使用Perl的Data::Dumper模块，它会把一个（可能自引用）的数据结构转换为可以在外部保存的字符串，以后可以利用eval或do重新构造这个结构：

```

use Data::Dumper;
$Data::Dumper::Purity = 1; # 因为%TV是自引用的
open (FILE, "> tvinfo.perldata") || die "can't open tvinfo: $!";
print FILE Data::Dumper->Dump([\%TV], ['*TV']);
close(FILE) || die "can't close tvinfo: $!";

```

以后另外一个程序（或者同一个程序）可以从文件读取：

```

open (FILE, "< tvinfo.perldata") || die "can't open tvinfo: $!";
undef $/; # 一次读取整个文件

```



```
eval <FILE>; # 重新创建%TV
die "can't recreate tv data from tvinfo.perldata: $@" if $@;
close(FILE) || die "can't close tvinfo: $!";
print $TV{simpsons}{members}[2]{age};
```

或者简单写为：

```
do "tvinfo.perldata" || die "can't recreate tvinfo: $! $@";
print $TV{simpsons}{members}[2]{age};
```

`Storable`也是一个标准模块，它会用一种非常快速的紧缩二进制格式写出数据结构。`Storable`模块还支持自动文件锁定（如果你的系统实现了`flock`函数），甚至有一些有趣的钩子，使得对象类可以处理自己的串行化。可以如下使用`Storable`保存同一个结构：

```
use Storable qw(lock_nstore);
lock_nstore(\%TV, "tvdata.storable");
```

下面把它恢复到一个变量中，其中包含所获取的散列的引用：

```
use Storable qw(lock_retrieve);
$TV_ref = lock_retrieve("tvdata.storable");
```

`Storable`另外还提供了一个`dclone`函数，这会创建一个多层数据结构的“深”拷贝，使用这个函数往往比你自己编写函数要容易。

```
use Storable qw(dclone);
%TV_copy = % { dclone \%TV };
```

关于使用`Data::Dumper`和`Storable`的其他技巧请参考相关的文档。

还有很多其他的解决方案，从紧缩的二进制格式（速度很快）到XML（互操作性非常好）都可以作为存储格式。YAML是一种很好的中间选择，可读性相当好。可以查看离你最近的CPAN镜像！

这一章开始有点意思了，因为我们将讨论软件设计的内容。要讨论好的软件设计，不能不谈到懒惰、急躁和傲慢，这也是好的软件设计的基础。

很多时候本该定义一个更高层次的抽象（可能只需要一个循环或子例程），我们却经常落入一个陷阱，总想使用剪切和粘贴<sup>注1</sup>。没错，有些人可能会走另一个极端，有时本该使用剪切和粘贴，他们却定义了大量更高层次的抽象<sup>注2</sup>。不过，总的来讲，大多数情况下我们都应当考虑使用更多的抽象，而不是减少抽象。

在这两个极端中间，对于使用多少抽象才合适，还有一些人的态度还算中立，不过在本该重用现有代码时，他们往往很冲动地写他们自己的抽象<sup>注3</sup>。只要你想这样做，务必坐下来，好好想一想，从长远来看，怎样做对你和你周围的人最有好处。如果你想把你的创造力都投入到代码中，为什么不让你所在的这个世界变得更美好呢？即使你的目的仅仅是希望程序成功，也需要确保这个程序能很好地适应相应的“生态环境”。

要完成生态可持续性编程，第一步很简单：就是不要乱丢“垃圾”。写一段代码时，要为这个代码提供它自己的命名空间，这样你的变量和函数就不会与其他人的变量/函数冲突，反之亦然。命名空间有点像你自己的家，在这里你可以随心所欲，房间里怎么乱都行，只要保证从外面看还不错。在Perl中，命名空间称为包（package）。包提供了基本构造模块，基于这些构造模块，可以构建更高层次的模块和类概念。

就像“家”的概念一样，“包”的概念也有一点模糊。包是独立于文件的。一个文件中可

注1： 这是伪懒惰的一种形式。

注2： 这是伪傲慢的一种形式。

注3： 你应该猜到了，这是一种伪急躁。不过如果你确定要另起炉灶，重新发明，至少应当发明一个更好的东西。



以有多个包，或者一个包可能跨多个文件，就像你的家可能只是一幢大楼中的一个小阁楼（如果你是一个穷困潦倒的艺术家），或者也可能由很多幢建筑组成（也许你正好是女王）。不过一般来讲，家通常就是一个建筑，而包通常也只是一个文件。如果想在一个文件中放一个包，为此Perl提供了一些特殊的帮助，条件是文件名要与包名相同，并使用扩展名`.pm`，这是“perl module”（perl模块）的缩写。模块（module）在Perl中是可重用的基本单位。实际上，要使用模块，需要使用`use`命令，这是一个编译器指令，可以控制从一个模块导入子例程和变量。到目前为止，你已经见过很多`use`的例子，所有这些都属于模块重用。

应当把你的模块放在Perl综合典藏网（Comprehensive Perl Archive Network）或CPAN上，因为也许别人会发现你的模块很有用。正是由于程序员们热心地在Perl社区分享他们的劳动成果，Perl才得以茁壮成长。很自然地，你也会在CPAN中找到别人上传的让其他人使用的模块。有关的详细内容请参见第19章和<http://www.cpan.org>。

在过去的20多年，计算机语言的设计呈现一种趋势，越来越让人陷入某种偏执。你编写每一个模块要像处于一种戒严状态。当然，确实某些封建文化中这样是可以的，不过并不是所有文化都是这样。例如，在Perl文化中，你不能进入别人的家，这是因为你没有得到邀请，而不是因为窗户上有栅栏<sup>注4</sup>。

这不是一本关于面向对象方法论的书，我们也不打算把你变成一个极度推崇面向对象方法的狂热分子（即使你确实想变成那样）。有很多关于这方面的书。Perl的面向对象设计哲学与Perl在其他方面的哲学是一致的：要在适当的场合使用面向对象设计，而避免在不合适的地方使用。决定权在你。

按照面向对象的说法，每个对象都属于一个分组，这称为一个类（class）。在Perl中，类和包以及模块的关系很紧密，初学者通常认为它们是可互换的。一般的类由一个模块实现，模块会定义一个与类同名的包。后面几章我们会详细解释所有这些内容。

使用`use`加载模块时，可以受益于直接软件重用。利用类，则可以得到间接软件重用的好处，即一个类通过继承使用另一个类。利用类还能得到更多：这是提供给另一个命名空间的一个清晰的接口。类中的所有一切都要间接访问，从而将类与外面的世界隔离开。

我们在第8章提到过，Perl中的面向对象编程是使用引用实现的，引用的指示对象知道它们属于哪个类。实际上，既然你已经了解引用，那么几乎所有关于对象的难题都已经很清楚了。按照钢琴家的说法，其余的内容就像“全在指尖上”。当然，你还要再做一些练习。

一个基本的“指法”练习就是要学习如何保护不同的代码块，让它们不会意外地破坏其他代码块的变量。每个代码块都属于一个特定的包，包确定了这个代码块中可用的变量和子

---

注4： 不过，如果你确实希望如此，Perl也提供了一些栅栏。参见第20章中的“处理不安全的代码”一节。



例程。Perl遇到一个代码块时，会把它编译为我们所称的当前包（current package）。初始的当前包名为“main”，不过你可以在任何时刻用package声明将当前包切换为另一个包。当前包确定了要使用哪个符号表来查找变量、子例程、I/O句柄和格式。

## 符号表

包的内容统称为符号表（symbol table）。符号表存储在一个散列中，这个散列与包同名，不过后面附加两个冒号。因此，main符号表的名字就是%main::。由于main正好是默认包，Perl提供了%::作为%main::的简写形式。

类似地，Red::Blue包的符号表名为%Red::Blue::。由于main符号表包含所有其他顶层符号表，包括它本身，所以%Red::Blue::也就是%main::Red::Blue::。

我们说一个符号表“包含”另一个符号表时，是指它包含另一个符号表的一个引用。由于main是顶层包，它包含它本身的一个引用，所以%main::就等同于%main::main::，也等同于%main::main::main::，依此类推，无穷无尽。如果你要编写代码遍历所有符号表，一定要注意检查这种特殊情况，这一点非常重要。

在一个符号表的散列中，每个键/值对将一个变量名映射到这个变量的值。键是符号标识符，值是相应的类型团。所以使用\*NAME 型团记法时，实际上就是在访问散列（其中包含当前包的符号表）中的一个值。下面两行代码的效果（几乎）是一样的：

```
*sym = *main::variable;  
*sym = $main::{"variable"};
```

第一种更高效一些，因为这会在编译时访问main符号表。如果之前不存在这样一个类型团，它还会根据这个名字创建一个新的类型团，而第二种形式做不到这一点。

由于包是一个散列，因此可以查看包的键，得到包的所有变量。因为散列的值是类型团，你可以用多种方法对它们解引用。可以尝试这样做：

```
foreach $symname (sort keys %main::) {  
    local *sym = $main::{$symname};  
    print "\$$symname is defined\n" if defined $sym;  
    print "\@$symname is nonnull\n" if @sym;  
    print "%$symname is nonnull\n" if %sym;  
}
```

由于所有包都可以通过main包（直接或间接地）访问，因此可以编写Perl代码访问程序中的每一个包变量。用V命令要求Perl调试工具转储所有变量时，就是在以这种方式访问所有包变量。需要说明，如果这样做，你看不到用my声明的变量，因为它们独立于包，不过可以看到用our声明的变量。有关内容参见第18章。

之前我们说过，除了main外，包中只存储标识符。这里有点小错误：实际上可以使用你想

要的任何字符串作为符号表散列中的键。不过如果你想直接使用一个非标识符，这就不是合法的Perl代码：

```
$!@#$$%      = 0;      # 不正确，语法错误。
${'!@#$$%'}   = 1;      # 正确，不过未限定。

${'main::!@#$$%'} = 2;    # 可以在字符串中限定。
print ${ $main::{'!@#$$%'} } # 正确，打印2!
```

对类型团的赋值是一个别名操作；也就是说，如果有以下赋值：

```
*dick = *richard;
```

那么可以通过标识符`richard`访问的变量、子例程、格式、文件和目录句柄也可以通过符号`dick`来访问。如果你只想对一个特定的变量或子例程指定别名，那么可以对一个引用赋值：

```
*dick = \ $richard;
```

这会使`$richard`和`$dick`变成同一个变量，不过`@richard`和`@dick`仍是不同的数组。很高深，是不是？

Exporter从一个包向另一个包导入符号时就是这样做的。例如，如果有以下赋值：

```
*SomePack::dick = \&OtherPack::richard;
```

这将从包`OtherPack`向`SomePack`包导入`&richard`函数，可以作为`&dick`函数来访问（Exporter模块将在下一章介绍）。如果在赋值前面加一个`local`，这个别名操作只会持续到当前动态作用域结束。

可以利用这个机制从子例程获取引用，使指示对象可以作为一个适当的数据类型：

```
*units = populate();      # 将\%newhash赋至类型团
print $units{kg};         # 打印70，不需要解引用！

sub populate {
    my %newhash = (km => 10, kg => 70);
    return \%newhash;
}
```

类似地，可以把一个引用传递到子例程，使用时不需要解引用：

```
%units = (miles => 6, stones => 11);
fillerup( \%units );      # 传入一个引用
print $units{quarts};     # 打印4

sub fillerup {
    local *hashsym = shift; # 将\%units赋至类型团
    $hashsym{quarts} = 4;   # 影响%units，不需要解引用！
}
```

这些都是比较巧妙的方法，如果你不想显式地解引用，可以采用这些方法以很小的开销传递引用。要注意，这两种技术只适用于包变量；如果我们用`my`来声明`%units`，这些技术就不起作用了。

符号表的另一种用法是建立“常量”标量：

```
*PI = \3.14159265358979;
```

现在你不能改变`$PI`，总的来说这可能是一件好事。这与常量子例程不同（常量子例程在编译时优化）。常量子例程定义有原型，要求不带参数，并返回一个常量表达式；有关的详细内容请参见第7章中的“内联常量函数”一节。`use constant pragma`（参见第29章）是一个很方便的简写形式：

```
use constant PI => 3.14159;
```

在底层，这会使用`*PI`的子例程槽，而不是之前使用的标量槽。这等价于以下更紧凑（但不太可读）的形式：

```
*PI = sub () { 3.14159 };
```

这是一个很方便的技术，一定要有所了解，将`sub {}`赋至一个类型团是在运行时为匿名子例程指定名字的一种方法。

将类型团引用赋至另一个类型团（`*sym = \*oldvar`）等同于赋整个类型团，因为Perl会自动对这个类型团引用解引用。将类型团设置为一个简单的字符串时，会得到用这个字符串命名的整个类型团，因为Perl会在当前符号表中查找这个字符串。下面的代码都是等价的，不过前两个会在编译时计算符号表记录，而后两个会在运行时计算：

```
*sym = *oldvar;  
*sym = \*oldvar;      # 自动解引用  
*sym = *{"oldvar"};   # 显式符号表查找  
*sym = "oldvar";      # 隐式符号表查找
```

完成以下赋值时，只是替换了类型团中的一个引用：

```
*sym = \ $frodo;  
*sym = \ @sam;  
*sym = \ %merry;  
*sym = \ &pippin;
```

如果换个角度考虑，类型团本身可以看作是一种散列，其中包含对应不同变量类型的记录。在这里，键是固定的，因为类型团只能包含一个标量、一个数组、一个散列等。不过，可以取出单个引用，如下所示：

```
*pkg::sym{SCALAR}    # 等同于\ $pkg::sym  
*pkg::sym{ARRAY}     # 等同于\ @pkg::sym  
*pkg::sym{HASH}      # 等同于\ %pkg::sym
```



```

*pkg::sym{CODE}      # 等同于\&pkg::sym
*pkg::sym{GLOB}      # 等同于\*pkg::sym
*pkg::sym{IO}        # 内部文件/目录句柄，没有直接的等价记法
*pkg::sym{NAME}      # "sym"（不是一个引用）
*pkg::sym{PACKAGE}   # "pkg"（不是一个引用）

```

可以用`*foo{PACKAGE}`和`*foo{NAME}`查找`*foo`符号表记录来自哪个名字和包。对于传入类型团作为参数的子例程来说，这可能很有用：

```

sub identify_typeglob {
    my $glob = shift;
    print "You gave me ", *{$glob}{PACKAGE}, "::", *{$glob}{NAME}, "\n";
}
identify_typeglob(*foo);
identify_typeglob(*bar::glarch);

```

这会打印以下结果：

```

You gave me main::foo
You gave me bar::glarch

```

可以用`*foo{THING}`记法来得到`*foo`的单个元素的引用。有关的详细内容参见第8章中的“符号表引用”一节。

这个语法主要用来得到内部文件句柄或目录句柄的引用，因为其他内部引用完全可以用其他方式来访问（原来`*foo{FILEHANDLE}`可以表示`*foo{IO}`，不过现在已经不支持这种记法）。但我们认为可以对它加以推广，因为这看起来很漂亮（可以这么说吧）。你可能并不需要记住所有这些，除非你打算另外编写一个Perl调试工具。

## 限定名

通过加前缀（prefixing）或限定（qualifying），也就是在标识符前面加包名和一个双冒号（`$Package::Variable`）前缀，可以指示其他包中的标识符<sup>注5</sup>。如果包名是`null`，则假设为`main`包。也就是说，`$::sail`等价于`$main::sail`<sup>注6</sup>。

原来的包定界符是一个单引号，所以在非常老的Perl程序中，你会看到类似`$main'sail` 和 `$somepack'horse`的变量。不过，现在理想的定界符通常是双冒号，部分原因是对人来说这样更具有可读性，另外一部分原因是这对于`emacs`宏也更可读。它还会让C++程序员感

---

注5：提到标识符，我们是指作为符号表键的名字，用来访问标量变量、数组变量、散列变量、子例程、文件或目录句柄，以及格式。从语法上来讲，标签也是标识符，不过它们并不放到特定的符号表中；实际上，它们会直接关联到程序中的语句。标签不能用包名限定。

注6：为了澄清另一个可能混淆的概念，在类似`$main::sail`的变量名中，我们把`main`和`sail`称为“标识符”，但`main::sail`合在一起不认为是标识符。我们把它称为一个变量名，因为标识符不能包含冒号。

觉他们很清楚自己在做什么，这与使用单引号作为分隔符不同，原来用单引号只会让Ada程序员感觉他们知道自己在做什么。因为老式的语法仍得到支持，以保证向后兼容，如果你想使用类似"This is \$owner's house"的字符串，实际上会访问\$owner::s；也就是包owner中的\$s变量，这可能并不是你想要的。可以使用大括号来消除二义性，如"This is \${owner}'s house"。

可以在包名中用双冒号将标识符串在一起：\$Red::Blue::var。这表示\$var属于Red::Blue包。Red::Blue包与Red或Blue包（如果恰好存在）没有任何关系。也就是说，Red::Blue和Red或Blue之间的关系可能对编写或使用这个程序的人有意义，但是对Perl来说没有任何意义（嗯，有一点除外，在当前实现中，符号表Red::Blue恰好存储在Red符号表中。不过Perl语言并没有直接利用这一点）。

很早以前，如果变量以一个下画线开头，会把它强制放在main包中，不过我们后来决定允许编写包的程序员使用一个前导下画线来指示半私有标识符（这表示只能在这个包内部使用），这样会更有用（真正的私有变量可以声明为文件作用域词法变量，不过这在包和模块存在一对一的关系时最适用，这种情况很常见，但并不是必要的）。

%SIG散列（用于捕获信号，参见第15章）也很特殊。如果将一个信号句柄定义为字符串，会假设它指示main包中的一个子例程，除非明确地使用了另一个包名。如果你想指定一个特定的包，则要使用一个完全限定的信号句柄名，或者通过赋一个类型团或函数引用完全避免使用字符串：

```
$SIG{QUIT} = "Pkg::quit_catcher"; # 完全限定句柄名
$SIG{QUIT} = "quit_catcher";      # 表示"main::quit_catcher"
$SIG{QUIT} = *quit_catcher;       # 强制当前包的子例程
$SIG{QUIT} = \&quit_catcher;      # 强制当前包的子例程
$SIG{QUIT} = sub { print "Caught SIGQUIT\n" }; # 匿名子例程
```

## 默认包

默认包是main，这与C中的顶层子例程名类似。除非另外指出（后面将会介绍），否则所有变量都在这个包中。下面的代码都是一样的：

```
#!/usr/bin/perl

$name      = 'Amelia';
$main::name = 'Amelia';

$type      = 'Camel';
$main::type = 'Camel';
```

如果使用strict，就必须对变量做出声明，因为这个pragma不允许使用未声明的变量：

```
#!/usr/bin/perl
use v5.12;
```



```
$name      = 'Amelia'; # 编译时错误
$main::name = 'Amelia';

$type      = 'Camel'; # 编译时错误
$main::type = 'Camel';
```

只有标识符（用字母或下画线开头的名字）存储在包的符号表中。所有其他符号都放在main包中，包括非字母数字变量，如\$!，\$?和\$<sub>注7</sub>。另外，如果未限定，会强制将标识符STDIN、STDOUT、STDERR、ARGV、ARGVOUT、ENV、INC和SIG放在包main中，即使其用途与相应的内置标识符不同时也是如此。不要将你的包命名为m、s、y、tr、q、qq、qr、qw或qx，除非你想自找麻烦。例如，不能使用标识符的限定形式作为文件句柄，因为这会解释为一个模式匹配、替换或转换。

## 改变包

“当前包”既是一个编译时概念也是一个运行时概念。大多数变量名查找都发生在编译时，不过对符号引用解引用时，以及在eval下编译新代码时，会发生运行时查找。具体地，用eval计算一个字符串时，Perl知道在哪个包中调用这个eval，并在计算这个字符串时传播这个包（当然，完全可以在eval字符串中切换到另外一个包，因为eval字符串会被看作是一个块，就像用do、require或use加载的文件一样）。

出于这个原因，每个package声明都必须声明一个完整的包名。任何包名都不能假设有某个隐含的“前缀”，即使（看上去）它是在另外某个包声明的作用域内声明的。

或者，如果一个eval希望找出它所在的包，特殊符号\_\_PACKAGE\_\_会包含当前包的名字。你可以把它当作一个字符串，所以可以在一个符号引用中用它来访问包变量。不过，如果这样做，很可能要用our声明变量，这样才能把它当作一个词法作用域变量来访问。

未用my声明的所有变量都与一个包关联，甚至类似\$\_和%SIG等看上去无所不在的变量也是如此。其他变量都使用当前包，除非加了限定：

```
$name = 'Amelia';           # 当前包中的名

$Animal::name = 'Camelia';  # Animal包中的名
```

package声明会改变作用域其余部分（块、文件或eval，要看哪一个最先出现）的默认包，或者直到出现同级的另一个包声明，它会覆盖之前的声明（这是一种常用的做法）：

```
package Animal;

$name = 'Camelia';  # $Animal::name
```

---

注7： 不过，在v5.10中可以有一个词法作用域变量\$<sub>注7</sub>。



要特别（反复）强调很重要的一点，`package`并不创建作用域，所以它不会隐藏相同作用域中的词法作用域变量：

```
my $type = 'Camel';

package Animal;

print "Type is $type\n";    # 词法作用域变量$type，因此为"Camel"
$type = 'Ram';

package Zoo;

print "Type is $type\n";    # 词法作用域变量$type，因此为"Ram"
```

要想优先使用同名变量的包版本作为相同作用域中的词法作用域变量，需要使用`our`。不过要当心。这会使当前包的变量成为作用域其余部分的默认变量，即使默认包改变也是如此：

```
my $type = 'Camel';

package Animal;

our $type = 'Ram';
print "Type is $type\n";    # 当前包$type，所以为"Ram"

package Zoo;

print "Type is $type\n";    # Animal $type，所以为"Ram"
```

在包Zoo中，`$type`仍是`$Animal::type`版本。`our`将应用于作用域的其余部分，而不是包声明的其余部分。这可能让人有些糊涂。要记住，`package`只会改变默认包名；它不会结束或开始作用域。一旦改变了包，所有后续的未声明的标识符都会放在当前包的符号表中。如此而已。

一般地，`package`声明往往是将由`require`或`use`加载的文件中的第一个语句。不过，重申一次，这只是一般约定。你完全可以把`package`声明放在任何地方，能放语句的地方都可以放`package`声明。甚至可以把它放在块的末尾，只是在这种情况下它没有任何作用。可以在多个不同地方切换包；包声明只是在为代码块的其余部分选择编译时使用哪个符号表。利用这种方法，一个给定的包可以跨多个文件。

从最近的版本开始，可以在包声明行中指定包的版本：

```
package Zoo v3.1.4;
```

另外，在v5.14及以后版本中还可以使用加括号的形式（看起来更像是标准代码块）。这会将包的作用域限制为仅在代码块内部。通过使用这个特性，可以避免之前提到的名字外溢问题：

```

my $type = 'Camel';

package Animal {
    our $type = 'Ram';
    print "Type is $type\n" ; # 当前包$type, 所以为"Ram"
}

package Zoo v3.1.4 {
    print "Type is $type\n"; # 外部$type, 所以为"Camel"
}

```

## 自动加载

正常情况下，不能调用一个尚未定义的子例程。不过，如果在未定义子例程的包中（或者对于一个对象方法，则是在该对象某个基类的包中）有一个名为AUTOLOAD的子例程，那么调用这个AUTOLOAD子例程时会为它提供传入原子例程的同样的参数。可以定义AUTOLOAD子例程来返回值（就像一个普通的子例程一样），或者也可以让它定义不存在的例程，然后调用该例程，就好像这个例程一直都存在一样。

原子例程的完全限定名会魔法般地出现在包全局变量\$AUTOLOAD中，与AUTOLOAD例程在同一个包中。

下面给出一个简单的例子，这里礼貌地警告你调用了未定义的子例程，而不是断然退出：

```

sub AUTOLOAD {
    our $AUTOLOAD;
    warn "Attempt to call $AUTOLOAD failed.\n";
}

blarg(10);          # $AUTOLOAD将设置为main::blarg
print "Still alive!\n";

```

或者，可以代表未定义子例程返回一个值：

```

sub AUTOLOAD {
    our $AUTOLOAD;
    return "I see $AUTOLOAD(@_)\n";
}

print blarg(20);    # 打印: I see main::blarg(20)

```

你的AUTOLOAD子例程可能会使用eval或require为未定义的子例程加载一个定义，或者使用之前讨论过的团赋值技巧，然后使用特殊形式的goto执行这个子例程，它可以消除AUTOLOAD例程的栈帧而没有跟踪记录。下面向团赋一个闭包来定义这个子例程：

```

sub AUTOLOAD {
    my $name = our $AUTOLOAD;
    *$AUTOLOAD = sub { print "I see $name(@_)\n" };
    goto &$AUTOLOAD; # 重启新例程
}

```

```

}

blarg(30);    # 打印: I see main::blarg(30)
glarb(40);    # 打印: I see main::glarb(40)
blarg(50);    # 打印: I see main::blarg(50)

```

模块编写者使用标准AutoSplit模块把他们的模块划分到单独的文件中（文件名以`.al`结尾），每个文件包含一个例程。这些文件放在系统Perl库的`auto/`目录中，在此之后，可以根据标准AutoLoader模块的需要自动加载这些文件。

SelfLoader模块采用了一种类似的方法，只不过它会从文件自己的DATA域自动加载函数，这样做在某些方面效率不高，但在另外一些方面可能更高效。利用AutoLoader和SelfLoader自动加载Perl函数，这类似于利用DynaLoader动态加载已编译C函数，只不过自动加载的粒度是函数调用，而动态加载的粒度是完整的模块，而且通常会一次链入多个C或C++函数（需要说明，很多Perl程序员没有使用AutoSplit、AutoLoader、SelfLoader或DynaLoader模块也能很好地完成工作。你只需要知道存在这些模块，如果某一天其他方法都不奏效时就可以考虑使用这些模块）。

利用AUTOLOAD例程可以做一些有意思的处理，把它作为其他接口的包装器。例如，假设未定义的函数要调用system并提供它的参数。只需要这样做：

```

sub AUTOLOAD {
    my $program = our $AUTOLOAD;
    $program =~ s/.*:://; # 截断包名
    system($program, @_);
}

```

祝贺你，现在你已经实现了一个作为Perl标准的基本Shell模块。可以如下调用你的自动加载工具（在UNIX上）：

```

date();
who("am", "i");
ls("-l");
echo("Abadugabudabuda...");

```

实际上，如果预声明希望以这种方式调用的函数，可以假装它们是内置函数，忽略调用中的小括号：

```

sub date ($$);           # 允许0到2个参数。
sub who ($$$$);          # 允许0到4个参数。
sub ls;                  # 允许任意数目的参数。
sub echo ($@);           # 允许至少一个参数。

date;
who "am", "i";
ls "-l";
echo "That's all, folks!";

```



在v5.8中，AUTOLOAD可以有一个:lvalue属性。

```
package Chameau;
use v5.14;

sub new { bless {}, $_[0] }

sub AUTOLOAD :lvalue {
    our $AUTOLOAD;
    my $method = $AUTOLOAD =~ s/.*:://r;
    $_[0]->{$method};
}

1;
```

采用这个方法，可以访问它或者为它赋值：

```
use v5.14;
use Chameau;

my $chameau = Chameau->new;

$chameau->awake = 'yes';

say $chameau->awake;
```

或者，可以让最后一个值变成一个符号引用：

```
package Trampeltier;

sub new { bless {}, $_[0] }
sub AUTOLOAD :lvalue { no strict 'refs'; *{$AUTOLOAD} }

1;
```

所以，可以通过为它赋值来定义方法：

```
use Trampeltier;

my $trampeltier = Trampeltier->new;

$trampeltier->name = sub { 'Amelia' };
```

不过，你可能并不想这么做。

模块是Perl中代码重用的基本单元。在底层，模块就是在同名文件（末尾要加`.pm`）中定义的一个包。这一章中，我们将探讨如何使用其他人的模块，以及如何创建你自己的模块。

Perl提供了数百个很有用的模块，可以在Perl安装目录的`lib`目录下找到这些模块，这个目录是你（或其他人）构建perl时确定的。用`-V`开关可以看到这些目录在哪里：

```
% perl -V
Summary of my perl5 (revision 5 version 14 subversion 1) configuration:

...
Built under darwin
Compiled at Jul 5 2011 21:43:59
@INC:
  /usr/local/perl/lib/site_perl/5.14.2/darwin-2level
  /usr/local/perl/lib/site_perl/5.14.2
  /usr/local/perl/lib/5.14.2/darwin-2level
  /usr/local/perl/lib/5.14.2
```

用`corelist`可以看到perl提供的所有模块，`corelist`也是perl提供的：

```
% corelist -v 5.014
```

所有标准模块还提供了丰富的在线文档，这些文档可能比这本书的内容更新。可以试着用`perldoc`命令阅读这些文档：

```
% perldoc Digest::MD5
```

Perl综合典藏网（Comprehensive Perl Archive Network，CPAN）包含了全球Perl社区贡献的模块存储库，这个内容将在第19章讨论。参见<http://www.cpan.org>。

# 加载模块

模块有两种类型：传统模块和面向对象模块。传统模块定义子例程和变量，供调用者导入和使用。面向对象模块相当于类定义，可以通过方法调用来访问，有关内容见第12章的介绍。有些模块则兼具这两类模块的特点。

Perl模块通常通过以下声明包含在程序中：

```
use MODULE;
```

这等价于：

```
BEGIN {  
    require MODULE;  
    MODULE->import();  
}
```

这发生在编译阶段，所以模块中的所有代码都在编译阶段运行。这通常没有问题，因为模块中的大多数代码都在子例程或方法中。有些模块可能会加载另外的模块、XS代码，以及其他代码组件。由于Perl在遇到use时会处理use，对@INC的所有修改都必须在use之前发生。你可能想使用lib pragma（见第29章），这也要用use加载。

如果希望在运行阶段加载模块，可能要延迟包含，直到运行一个真正需要这个模块的子例程时才包含该模块，这种情况下可以使用require：

```
require MODULE;
```

*MODULE*必须是一个包名，它将转换为模块文件。use会把::转换为/，然后在末尾追加一个.pm。它将在@INC中查找这个名字。如果你的模块名为Animal::Mammal::HoneyBadger，就会查找Animal/Mammal/HoneyBadger.pm。一旦加载，Perl找到的文件路径就会出现在%INC中。Perl一次加载一个文件。在它尝试加载一个文件之前，会在%INC中查看是否已经加载这个文件。如果已经加载，可以重用之前加载的结果。

可以用require直接加载文件，要使用正确的路径分隔符（这可能不是可移植的）：

```
require FILE;  
require 'Animal/Mammal/HoneyBadger.pm';
```

不过，一般来讲，我们更倾向于使用use而不是require，因为它会在编译时查找模块，这样你能更快地发现错误。

有些模块在其导入列表中提供了额外的功能。这个列表将成为import的参数表：

```
use MODULE LIST;
```

如：



```
BEGIN {
  require MODULE;
  MODULE->import(LIST);
}
```

模块的import可以做它想做的任何事情，不过大多数模块仍使用从Exporter继承的import版本，这个内容我们稍后介绍。一般地，import把符号（子例程和变量）放在当前命名空间中，从而使编译单元的其余部分可以访问这些符号。有些模块有一个默认的导入列表。

例如，Hash::Util模块会导出多个符号来完成特殊的散列动作。use引入lock\_keys符号，允许编译单元的其余部分访问：

```
use Hash::Util qw(lock_keys);

lock_keys( my %hash, qw(name location) );
```

即使没有LIST，它也可以根据模块的默认列表导入一些符号<sup>注1</sup>。File::Basename模块会自动导入一个basename、dirname和fileparse：

```
use File::Basename;

say basename($ARGV[0]);
```

如果你根本不希望任何导入，可以显式地提供一个空列表：

```
use MODULE ();
```

有时你可能想使用一个模块的某个特定版本（或更新的版本），通常这是为了避免之前版本中已知的问题，或者想使用一个不向后兼容的更新的API：

```
use MODULE VERSION LIST;
```

正常情况下，任何大于或等于VERSION的版本都是可以的。不能指定某一个版本或一个版本范围。不过，模块可以做不同的处理，因为实际上要由VERSION方法确定做什么。

## 上传模块

use反过来就是no。no不调用import，而是调用unimport。这个方法可以做它想做的任何事情。语法是一样的：

```
no MODULE;
no MODULE LIST;
no MODULE VERSION;
no MODULE VERSION LIST;
```

---

注1： 如今这往往认为是不礼貌的。应当让人们指定他们想要什么，这样有助于避免两个不同模块导入相同符号时出现冲突。

你可能希望某些符号只在短时间内可用。例如，Moose模块是在Perl的内置特性之上构建的一个对象系统，可以导入很多便利方法。has方法会声明属性，不过一旦用完这些名字，就不需要保留。可以在需要Moose模块的代码段末尾用no取消导入：

```
package Person;
use Moose;

has "first_name" => (is => "rw", isa => "Str");
has "last_name" => (is => "rw", isa => "Str");

sub full_name {
    my $self = shift;
    $self->first_name . " " . $self->last_name
}

no Moose; #从Person包中删除关键字
```

要临时关闭一个strict特性，可以取消导入这个碍事的特性。应当使用尽可能小的作用域，以避免漏掉其他问题：

```
my $value = do {
    no strict "refs";

    ${ "${class}::name" }; # 符号引用
};
```

类似地，你可能需要临时关闭某种类型的警告，所以可以取消导入这种类型的警告：

```
use warnings;
{
    no warnings 'redefine';
    local *badger = sub { ... };
    ...;
}
```

## 创建模块

这一章中，我们只介绍模块的代码部分。要创建一个发行版本（distribution），还有很多工作要做，这个内容将在第19章介绍。

## 命名模块

一个好名字是创建模块很重要的一个部分。一旦选择了名字，而且一旦人们开始使用你的模块，你就必须一直坚持使用这个名字，因为用户会拒绝更新他们的代码。如果要把你的模块上传到CPAN，你肯定也希望人们能很容易地找到它。可以了解一下PAUSE中提供的命名原则。

除非作为pragma，否则模块名应当首字母大写。pragma（见第29章）实际上是编译器指令（对编译器的提示），所以我们保留了小写的pragma名，以供将来使用。

如果你想建立一个私有模块，希望它的名字永远不会与标准库中或CPAN上的某个模块冲突，可以使用Local命名空间。CPAN没有禁止这样做，不过按惯例不会使用。

## 示例模块

之前我们说过，有两种类型的模块：传统模块或面向对象模块。下面将分别给出这两类模块最简单的例子。

面向对象模块很容易展示，因为面向对象模块与其用户通信并不需要太多“基础设施”。所有一切都通过方法来完成：

```
package Bestiary::OO 1.001;

sub new {
    my( $class, @args ) = @_ ;
    bless {}, $class;
}

sub camel { "one-hump dromedary" }
sub weight { 1024 }

### 这里还有更多方法

1;
```

使用这个模块的程序可以利用方法来完成所有工作：

```
use v5.10;
use Bestiary::OO;

my $bestiary = Bestiary::OO->new; # 类方法

say "Animal is ", $bestiary->camel(),
    " has weight ", $bestiary->weight();
```

要构造一个名为Bestiary的传统模块，需要创建一个*Bestiary.pm*文件，如下所示：

```
package Bestiary 1.001;
use parent qw(Exporter);

our @EXPORT = qw(camel);      # 默认导出的符号
our @EXPORT_OK = qw($weight); # 根据请求导出的符号

### 这里包含你的变量和函数

sub camel { "one-hump dromedary" }

$weight = 1024;
```



```
1; # 以一个表达式结尾，它计算为true
```

现在程序可以声明`use Bestiary`，这样能访问`camel`函数（但不能访问`$weight`变量），另外可以声明`use Bestiary qw(camel $weight)`，这样一来，函数和变量都可以访问：

```
use v5.10;

use Bestiary qw(camel $weight);

say "Animal is ", camel(), "has weight $weight";
```

还可以创建模块动态地加载用C编写的代码，不过这个内容不在这里介绍。

## 模块私有性和Exporter

Perl不会自动在其模块的私有/公共边界上检查，这与C++、Java和Ada等语言不同，Perl并不因强制私有性受到困扰。Perl模块可能希望你远离它的卧室，这是因为你没有得到邀请，而不是因为它有枪。

模块与用户有一个合约，其中一部分是习惯性合约，另外一部分则是书面合约。习惯性合约部分指定禁止模块改变任何未要求它改变的命名空间。模块的书面合约（也就是文档）可以指定其他限制性条款。不过，一旦你读过合约，则认为你已经很清楚声明`use RedefineTheWorld`时就是要重新定义这个世界，你要承担相应的后果和可能的风险。要重新定义这个世界，最常用的方法是使用`Exporter`模块。另外在这一章后面会看到，甚至可以用这个模块重新定义内置函数。

用`use`加载一个模块时，这个模块通常会提供一些变量或函数允许你的程序访问，或者更确切地讲，允许程序的当前包使用这些变量和函数。这种从模块导出符号的行为（相应地导入到你的程序中）有时被称为“污染”（polluting）你的命名空间。大多数模块都用`Exporter`来完成这个工作。正是由于这个原因，大多数模块中最前面的位置都有类似这样的声明：

```
use parent qw(Exporter);

require Exporter;
our @ISA = ("Exporter");
```

基于这两行声明，模块将继承`Exporter`类。继承的有关内容将在下一章介绍，不过现在你只需要知道：利用类似下面的代码，现在我们的`Bestiary`模块可以把符号导出到其他包中：

```
our @EXPORT      = qw($camel %wolf ram);          # 默认导出
our @EXPORT_OK   = qw(leopard @llama $emu);       # 根据请求导出
our %EXPORT_TAGS = (                              # 作为组导出
    camelids => [qw($camel @llama)],
    critters => [qw(ram $camel %wolf)],
);
```

从导出模块的角度来看，@EXPORT数组包含将默认导出的变量和函数的名字也就是程序声明 `use Bestiary` 时得到的变量和函数。另外，只有当程序在 `use` 语句中特别做出请求时，才会导出@EXPORT\_OK中的变量和函数。最后，%EXPORT\_TAGS中的键/值对允许程序包含@EXPORT和@EXPORT\_OK中所列的特定符号组。

从导入包的角度看，`use` 语句指定了一个要导入的符号列表、%EXPORT\_TAGS中指定的一个组、一个符号模式，或者什么都没有（在这种情况下，将从模块向程序中导入@EXPORT中的符号）。

可以包含以下任意语句，从Bestiary模块导入符号：

```
use Bestiary;           # 导入@EXPORT符号
use Bestiary ();        # 什么都不导入
use Bestiary qw(ram @llama); # 导入ram函数和@llama数组
use Bestiary qw(:camelids); # 导入$camel和@llama
use Bestiary qw(:DEFAULT); # 导入@EXPORT符号
use Bestiary qw(/am/);    # 导入$camel, @llama和ram
use Bestiary qw(/^$/);    # 导入所有标量
use Bestiary qw(:critters !ram); # 导入critters, 但不包括ram
use Bestiary qw(:critters !:camelids); # 导入critters, 但不包括camelids
```

如果从导出列表中去掉某个符号（或者用感叹号显式地将它从导入列表删除），使用这个模块的程序并不会因此无法访问这个符号。通过完全限定包名，如%Bestiary::gecko，程序总是可以访问模块的包的内容（由于词法作用域变量不属于包，私有性仍能保证，参见下一章中的“私有方法”一节）。

可以使用 `BEGIN { $Exporter::Verbose=1 }` 查看如何处理规范，另外向包中具体导入了哪些内容。

Exporter本身是一个Perl模块，如果你很好奇它是如何工作的，可以看到它使用了类型团技巧将符号从一个包导出到另一个包。在Exporter模块中，关键的函数是import，它会完成必要的别名处理，使一个包中的符号出现在另一个包中。实际上，`use Bestiary LIST` 语句就等价于：

```
BEGIN {
    require Bestiary;
    import Bestiary LIST;
}
```

这意味着，你的模块并不一定非得使用Exporter。使用Exporter时，模块可以做它喜欢的任何事情，因为use只是调用了这个模块的import方法，而你可以定义这个import方法来完成你想做的任何事情。

## 不使用Exporter的import方法实现导出

Exporter定义了一个名为`export_to_level`的方法，如果（出于某种原因）不能直接调用Exporter的import方法，就要使用这个方法。`export_to_level`方法可以如下调用：

```
MODULE->export_to_level($where_to_export, @what_to_export);
```

这里`$where_to_export`是一个整数，指示要把符号导出到调用栈中的哪一层调用程序，`@what_to_export`是一个数组，列出了要导出的符号（通常是@\_）。

例如，假设Bestiary有一个自己的import函数：

```
package Bestiary;
@ISA = qw(Exporter);
@EXPORT_OK = qw ($zoo);

sub import {
    $Bestiary::zoo = "menagerie";
}
```

由于有这个import函数，因此不会继承Exporter的import函数。如果希望一旦设置`$Bestiary::zoo`，Bestiary的import函数要表现得像Exporter的import函数一样，可以如下定义：

```
sub import {
    $Bestiary::zoo = "menagerie";
    Bestiary->export_to_level(1, @_);
}
```

这会把符号导出到当前包“上面”一层的包中。也就是说，导出到使用Bestiary的程序或模块中。

不过，如果这就是你想要的，大可不必从Exporter继承，完全可以导入import方法：

```
package Bestiary;
use Exporter qw(import); # v5.8.3及以后版本
```

## 版本检查

如果你的模块定义了一个`$VERSION`变量，使用模块的程序可以确保这个模块足够新。例如：

```
use Bestiary 3.14;    # Bestiary必须是3.14或更新版本
use Bestiary v1.0.4;  # Bestiary必须是1.0.4或更新版本
```

这些会转换为对`Bestiary->VERSION`的调用，这是从UNIVERSAL继承的（参见第12章）。

如果使用require，仍然可以直接调用VERSION来检查版本：

```
require Bestiary;
```



```
Bestiary->VERSION( '2.71828' );
```

目前，模块版本有些过于复杂，其中一些不必要的复杂性是由于一些无法避免的历史原因造成的。版本最初是Perl在模块包的\$VERSION中找到的版本。这可能是一个数字、一个字符串或者可能是某个操作的结果。多年来，对于版本字符串并没有标准化，所以人们可能会给出“1.23alpha”<sup>注2</sup>之类奇怪的版本。下面的版本都是一样的：

```
our $VERSION = 1.002003;
our $VERSION = '1.002003';
our $VERSION = v1.2.3;

use version;
our $VERSION = version->new( "v1.2.3" );
```

原先这并没有什么问题，不过后来Perl在5.005和v5.6之间改变了它自己的版本号机制。现在有了一个*v-string*，这是一个表示版本的特殊的直接量，可以根据你的需求包含任意多个点号。这些v-string实际上是压缩为字符的整数。再后来，Perl建立了版本对象和version模块的概念。如果你要支持那些古老版本的Perl（遗憾地告诉你，v5.6也已经是个老古董了），最好用简单字符串。

Perl假设小数点后面的部分是3位，这样一来，比较版本时就显得很奇怪。版本1.9排在版本1.10前面，尽管按字典顺序排序时.9排在.1后面。Perl把它们分别看作是1.009和1.010。你没有必要非得喜欢这样做，但起码必须接受这种做法（不过，还是尽量使用v1.9形式，因为这样将来可以兼容）。

除了这些，对于正在开发的未发布的版本也有一个约定。如果在版本中加一个\_或-TRIAL，很多CPAN工具就会知道这不是一个稳定的版本。这样一来，作者可以向CPAN上传他们编写的模块，同时允许CPAN测试人员和预发行版本用户测试模块，而不要求所有其他人都使用这个可能有问题的版本（参见第19章）。

```
our $VERSION = '1.234_001';
```

必须有引号，下画线才能保留，否则会将其解析，因为编译器允许它们出现在数值直接量中。

DaidGoiden在"Versionnumbers shouldbe boring" (<http://www.dagolden.com/index.php/369/version-numbers-should-be-boring/>)中介绍了有关的更多内容。

需要说明，在新版本的Perl中，可以完全去掉our声明，直接写为：

```
package Bestiary v1.2.3;
```

---

注2： 要查看这样一些奇怪的版本，可以参阅CPAN:DistnameInfo模块识别版本所做的工作。

## 管理未知符号

有时，你可能希望避免某些符号的导出。一般地，如果模块中包含的一些函数或常量在某些系统上没有意义，就会出现这种情况。可以把它们放在`@EXPORT_FAIL`数组中，避免Exporter导出这些符号。

如果程序试图导入这样一些符号，Exporter会为模块提供一个机会，让它在生成错误之前以某种方式做出响应。为此要调用一个`export_fail`方法，并为它提供一个失败符号列表，可以如下定义`export_fail`方法（假设你的模块使用了Carp模块）：

```
use Carp;
sub export_fail {
    my $class = shift;
    carp "Sorry, these symbols are unavailable: @_";
    return @_;
}
```

Exporter提供了一个默认的`export_fail`方法，它只是不加修改地返回这个列表，并使use加载失败，同时对每个符号产生一个异常。如果`export_fail`返回一个空列表，没有记录任何错误，那么请求的所有符号都会导出。

## 标记处理工具函数

由于`%EXPORT_TAGS`中列出的符号也必须出现在`@EXPORT`或`@EXPORT_OK`中，Exporter提供了两个函数，允许你增加这些作为标记的符号集：

```
%EXPORT_TAGS = (foo => [qw(aa bb cc)], bar => [qw(aa cc dd)]);

Exporter::export_tags("foo");      # 将aa、bb和cc增加到@EXPORT
Exporter::export_ok_tags("bar");    # 将aa、cc和dd增加到@EXPORT_OK
```

如果指定的名字不是标记，则会出错。

## 覆盖内置函数

很多内置函数都可以覆盖（overridden），不过（就像在墙上打洞一样），只能在有充分理由的情况下偶尔为之。一般地，如果一个包试图在非UNIX系统上模拟缺少的内置功能，可能会覆盖内置函数（不要把覆盖与重载混淆，重载是为内置操作符增加额外的面向对象含义，而没有覆盖任何内容。有关的更多信息参见第13章中对重载方法的讨论）。

可以从一个模块导入名字来实现覆盖，平常的预声明不够好。更准确地讲，将一个代码引用赋值到类型团时会触发覆盖，如`*open = \&myopen`。另外，赋值必须出现在另一个包中；这样一来，就很难通过有意地利用类型团别名来制造意外的覆盖。不过，如果确实想要实现覆盖，不用灰心，因为`subs pragma`允许通过导入语法预声明子例程，这些名字将覆盖内置的名字：

```

use subs qw(chdir chroot chmod chown);
chdir $somewhere;
sub chdir { ... }

```

一般地，模块不应该把open或chdir之类的内置名字放在其默认的@EXPORT列表中导出，因为这些名字可能会不知不觉地出现在其他人的命名空间，并在人们不知情的情况下意外地改变语义。如果模块包含@EXPORT\_OK列表中的名字，导入者必须显式地请求覆盖这个内置名字，这样才能保证所有人都是可信的。

可以通过CORE伪包访问原版本的内置函数。因此，CORE::chdir总是原先编译到Perl的chdir版本，即使chdir关键字已经被覆盖。

确切地讲，几乎总是这样。前面覆盖内置函数的机制总是有意地限制在请求导入的包中。不过，还有一种更彻底的机制，如果希望允许在任何地方覆盖某个内置函数，可以使用这样一个机制，而不用考虑命名空间边界。这是通过在CORE::GLOBAL伪包中定义函数做到的。下面给出一个例子，这里将glob操作符替换为某个理解正则表达式的操作符（需要说明，要想清晰地覆盖Perl的内置glob，需要做的工作有很多，这个例子并没有实现所有这些工作，取决于出现在一个标量上下文还是列表上下文，glob的表现是不同的。实际上，很多Perl内置函数都有这种上下文敏感的行为，写得好的覆盖应当充分支持这些区分不同上下文的行为。如果想了解一个功能完备的glob覆盖的例子，可以学习Perl提供的File::Glob模块）。不过，下面仍给出一个不太全面的版本：

```

*CORE::GLOBAL::glob = sub {
    my $pat = shift;
    my @got;
    local *D;
    if (opendir D, ".") {
        @got = grep /$pat/, readdir D;
        closedir D;
    }
    return @got;
}

package Whatever;

print <^[a-z_]+\..pm\>; # 显示当前目录中的所有pragma

```

通过全局覆盖glob，这样会强制每一个命名空间中的glob操作符都有一个新的（而且是破坏性的）行为，而无需这些命名空间中模块的认可和协助。当然，这样做必须格外小心，如果非得这样做的话。实际上，通常并不需要全局覆盖。

对于覆盖，我们的哲学是：做一个重要的人固然很好，但更重要的是做一个好人。



# 对象

在学习本章内容之前，首先你要了解包和模块（参见第10章和第11章）。另外还需要知道引用和数据结构（参见第8章和第9章）。如果对面向对象编程（OOP）有所了解也很有帮助，所以下一节我们先对OOL（Object-Oriented Lingo，面向对象术语）做一个简单的回顾。

Perl的面向对象模型与你在其他语言中用到的面向对象模型可能有很大区别。所以学习这一章的内容时，最好先忘掉你从其他语言那里了解到的面向对象知识。

## 面向对象术语简单回顾

作为一个数据结构，对象（object）有自己的一组行为。我们通常称这些行为是对象的直接动作，有时会把对象拟人化。例如，我们可以说一个矩形“知道”如何在屏幕上显示，或者它“知道”如何计算它自己的面积。

通过作为一个类（class）的实例（instance），每个对象都有自己的行为。这个类会定义一些方法（methods）：也就是应用于这个类及其实例的行为。如果需要区别这些方法，我们把只应用于特定对象的方法称为实例方法（instance methods），而应用于整个类的方法称为类方法（class methods）。不过，这只是一个惯例，对Perl来说，方法就是方法，仅由其第一个参数的类型来区分。

可以把实例方法看作是由特定对象完成的某个动作，如打印、自行复制或者改变一个或多个属性（“将这把宝剑取名为Andúril”）。类方法可以同时作用于多个对象（“显示所有宝剑”），或者可以提供其他不依赖于特定对象的操作（“从现在开始，只要新铸一把宝剑，就要在这个数据库中注册它的主人”）。有些方法用来生成一个类的实例（对象），这些方法称为构造函数（“铸一把镶嵌宝石而且刻有秘密文字的宝剑”）。这些构造函数

通常是类方法（“为我造一把新宝剑”），不过也可能是实例方法（“造一把和这同样的宝剑”）。

一个类可以从父类（parent classes）继承（inherit）方法，父类也称为基类（base classes）或超类（superclasses）。如果一个类继承了另一个类，则称它是一个派生类（derived class）或子类（subclass）。有些文献用“基类”表示“最上层”超类，这让情况变得更加复杂。我们所说的基类没有这个含义。通过继承，新类可以与已有的类有相似的行为，同时允许修改父类的行为或者增加父类中没有的行为。如果调用一个方法，而类中没有这个方法的定义，Perl就会自动地在其父类中查找方法定义。例如，一个sword（宝剑）类可能从一个通用的blade（刀锋）类继承了attack（攻击）方法。父类本身也可以有父类，必要时Perl还会搜索这些更高层次的父类。blade类可能进一步从一个更通用的weapon（武器）类继承它的attack方法。

在一个对象上调用attack方法时，具体的行为取决于这个对象是一把宝剑还是一把弓箭。可能根本没有任何区别。如果宝剑和弓箭都是从通用的武器类继承其攻击行为，那么二者就没有区别。不过，如果不同类的行为不同，方法分派机制总会选择最适合当前对象的attack方法。总是为特定类型的对象选择最适合的行为是一个很有用的特性，也称为多态（polymorphism）。多态是“不在乎”的一种重要形式。

实现一个类时，必须关心对象的内部组成，不过使用一个类时，则应当把对象看作是一个黑盒子。你看不到里面有什么，也不需要知道它是如何工作的，你只是根据这个盒子上的说明与它交互，也就是通过类提供的方法进行交互。即使你确实知道这些方法会对对象做些什么，也一定要克制自己不要介入其中。这就像电视的遥控器：尽管你知道它在做什么，但是如果没有充分的理由，就不要去随便摆弄它的内部器件。

Perl允许你在需要时从类外部查看对象的内部。不过，这样做会破坏它的封装（encapsulation），封装原则要求对对象的所有访问都只能通过方法来完成。封装使发布的接口（即应当如何使用对象）与对象的实现（具体如何工作）解耦合。除了设计者与用户之间的这个不成文的契约，Perl并没有提供一个显式的接口设施。设计者和用户双方希望达成共识，建立行为准则，为此用户只依赖于这些有文档的接口，而设计者不会破坏这个接口。

Perl并不强制你采用某种特定的编程风格，也不像其他一些面向对象语言那样受到私有性的困扰。不过，Perl在自由性方面确实有些困惑。作为一个Perl程序员，你有一项自由：可以根据需要选择或多或少的私有性。实际上，Perl的类和对象中可以有比C++更强的私有性。也就是说，Perl不限制你做任何事情，具体来讲，它不会限制你约束自己，如果你有这种打算。本章后面的“私有方法”和“闭包作为对象”小节将展示如何增强你的原则性。

必须承认，关于对象远不止这些内容，还可以利用很多其他方式更多地了解面向对象设计。不过，这不是我们现在的目标。所以，继续我们的话题吧。



# Perl的对象系统

Perl没有为对象、类或方法的定义提供任何特殊的语法。实际上，它重用了现有的结构来实现这3个概念<sup>注1</sup>。这里有一些非常简单的定义，可以让你安心一些：

**对象就是引用 (reference) ……嗯，也就是指示对象 (referent)**

引用允许单个标量表示更大的数据集合，所以用引用来表示对象应该也不会让人奇怪。理论上讲，对象并不是严格意义上的引用，实际上它是引用所指示的指示对象。不过Perl程序员经常并不严格区分这二者的差别，另外因为我们觉得这是一个不错的转喻，所以会在合适的场合延用这种用法<sup>注2</sup>。

**类就是包**

包可用作类，可以使用包的子例程来执行类的方法，或者使用包的变量保存类的全局数据。通常会用一个模块维护一个或多个类。

**方法就是子例程**

对于一个用作类的包，只需要在包中声明子例程，这些子例程就会作为这个类的方法。方法调用是调用子例程的一种新方式，会传递一个额外的参数，也就是用来调用这个方法的对象或包。

## 方法调用

如果要用一个词浓缩面向对象编程的精华，那就是抽象 (abstraction)。热衷于面向对象的人整天总把一些高深字眼挂在嘴边，比如多态 (polymorphism)、继承 (inheritance) 和封装 (encapsulation)，而抽象正是所有这些华丽辞藻的基础。我们并不怀疑这些有趣的词语，不过会从实用的角度来考虑它们对于调用方法意味着什么。方法是对象系统的核心，因为方法提供了实现所有这些有趣词语所需的抽象层。不是直接访问对象中的某些数据，而需要调用实例方法来访问。不是直接调用某个包中的一个子例程，而需要调用一个类方法。通过在类的使用和类的实现之间引入这样一个间接层，程序设计者可以自由地修改类内部的实现，而使用这个类的程序并不会因此被破坏。

Perl调用方法时支持两种不同的语法形式。第一种是我们已经很熟悉的一种方式，在Perl中随处可见；第二种形式你可能在其他编程语言中看到过。不论采用哪种形式的方法调用，总是会向作为方法的子例程传入一个额外的初始参数。如果用一个类调用方法，参数就是这个类的名字。如果用一个对象来调用这个方法，参数则是这个对象的引用。不论这个参数是什么，我们都把它称为方法的调用者 (invocant)。对于一个类方法，调用者是包名。对于一个实例方法，调用者则是指定某个对象的引用。

---

注1： 这正是在向你展示一个软件重用的例子！

注2： 我们更喜欢语言的灵动性，而不是数学的严格性。不论你是否同意。



换句话说，就是要用这个调用者来调用方法。有些面向对象文献把它称为方法的代理（agent）或方法的作用物（actor）。从语法上讲，调用者既不是动作的主体，也不是动作的接收者。这更像是一个间接对象，代表完成动作的受益者，就像命令“为我铸一把宝剑”中的“我”。从语义上讲，可以认为调用者是调用方或被调用方，这要看你认为哪一个更合适。至于具体如何考虑，我们不会告诉你（嗯，这可不是我们要介绍的内容）。

大多数方法都会显式调用，不过也可能隐式调用，由对象析构函数、重载操作符或绑定变量触发时就会隐式调用方法。更确切地讲，这些并不是常规的子例程调用，而是由Perl代表对象自动触发的方法调用。析构函数将在这一章后面介绍，重载将在第13章介绍，另外第14章会介绍绑定。

方法与常规子例程有一个差别，表现在何时解析它们的包，也就是Perl何时（早或晚）确定为方法或子例程执行哪个代码。子例程的包在编译时解析，也就是开始运行程序之前<sup>注3</sup>。相比之下，在真正调用方法之前，都不会解析这个方法的包（原型会在编译时检查，也正是因为这个原因，常规子例程可以使用原型，而方法不能）。

不能更早些解析方法的包，其原因相当简单：包由调用者的类来确定，而在具体调用方法之前并不知道调用者是什么。面向对象的核心就是这样一个简单的逻辑链：一旦知道调用者，就会知道调用者的类，而一旦知道类，就能知道类的继承关系，一旦知道类的继承关系，就会知道要调用哪一个具体的子例程。

这个抽象逻辑是有代价的。由于方法采用后解析，所以Perl中的面向对象解决方案比相应的非面向对象解决方案运行得慢。相比后面要介绍的一些更有意思的技术，这可能慢得多。不过，很多常见问题并不是靠处理得更快来解决，而是要处理得更巧妙。这正是面向对象的闪光之处。

## 使用箭头操作符的方法调用

我们提到过，有两种不同的方法调用方式。第一种调用方法的方式如下：

```
INVOCANT->METHOD(LIST)
INVOCANT->METHOD
```

这种方式通常称为箭头调用形式（原因显而易见）。不要把->与=>混淆，后者相当于一个神奇的逗号。如果有参数，就必须有小括号。执行时，方法调用首先找到子例程，这由INVOCANT的类和METHOD名共同确定，然后调用这个子例程，并传入INVOCANT作为它的一个参数。

---

注3：更准确地讲，子例程会解析为一个特定的类型团，其引用将填入已编译的操作码（opcode）树。这个类型团的含义甚至可以在运行时协商，正是因为这个原因，AUTOLOAD可以为你自动加载一个子例程。不过，正常情况下，类型团的含义还会在编译时根据一个适当命名的子例程的定义来解析。

*INVOCANT*是一个引用时，我们说*METHOD*作为一个实例方法来调用；如果*INVOCANT*是一个包名，我们说*METHOD*作为一个类方法来调用。实际上这二者之间并没有差别，只不过包名与类本身有更明显的关联，而不是与类的对象关联。一定要知道，对象也知道自己的类。稍后会告诉你如何将一个对象与类名关联，不过即使不知道这一点也完全可以使用对象。

例如，要使用类方法*summon*构造一个对象，然后在得到的对象上调用实例方法*speak*，代码可以写为：

```
$mage = Wizard->summon("Gandalf");    # 类方法
$mage->speak("friend");                # 实例方法
```

*summon*和*speak*方法由*Wizard*类定义—或者由它继承的某个类定义。不过你不用担心这一点。不用介入*Wizard*的“家事”。

由于箭头操作符是左结合的（参见第3章），所以甚至可以把这两个语句合并为一条语句：

```
Wizard->summon("Gandalf")->speak("friend");
```

有时你想调用一个方法，但是预先不知道方法名。可以使用箭头形式的方法调用，将方法名替换为一个简单的标量变量：

```
$method = "summon";
$mage = Wizard->$method("Gandalf");    # 调用Wizard->summon

$travel = $companion eq "Shadowfax" ? "ride" : "walk";
$mage->$travel("seven leagues");      # 调用$mage->ride或$mage->walk
```

尽管这里使用方法名来间接地调用方法，不过`use strict 'refs'`并不禁止你这样使用，因为所有方法调用实际上都会在解析时以符号形式查找。

在我们的例子中，我们把一个子例程的名字保存在*\$travel*中，不过你也可以存储一个子例程引用。这样会绕过方法查找算法，不过有时这正是你想要的。参见本章后面“私有方法”一节，以及“UNIVERSAL：终极祖先类”一节中关于*can*方法的讨论。对于在某个特定实例上调用的特定方法，要创建这样一个方法的引用，参见第8章中“闭包”一节。

## 使用间接对象的方法调用

第二种形式的方法调用如下所示：

```
METHOD INVOCANT (LIST)
METHOD INVOCANT LIST
METHOD INVOCANT
```

*LIST*两边的小括号是可选的：如果忽略小括号，这个方法就相当于一个列表操作符。所以可以有类似下面的语句，所有这些语句都使用这种形式的方法调用。注意类名或实例后面



没有分号：

```
no feature "switch"; # 出于某种原因（见下面）
$mage = summon Wizard "Gandalf";
$nemesis = summon Balrog home => "Moria", weapon => "whip";
move $nemesis "bridge";
speak $mage "You cannot pass";
break $staff;          # 更安全的用法：break $staff ();
```

这个列表操作符语法对你来说应该很熟悉了；这与向`print`或`printf`传递文件句柄的方式是一样的：

```
print STDERR "help!!!\n";
```

这与英语句子“Give Gollum the Preciousss”也类似，所以我们把它称为间接对象（indirect object）形式。调用者在间接对象槽（indirect object slot）中。如果你读到将`system`或`exec`之类的内置函数传入它的“间接对象槽”中，这就表示你在提供一个额外的无逗号的参数，使用间接对象语法调用一个方法时也会在同样的位置上提供这样一个参数。

这种间接对象形式甚至允许你指定`INVOCANT`是一个`BLOCK`，这个`BLOCK`计算为一个对象（引用）或类（包）。这就允许你将这两个调用合并为一个语句，如下所示：

```
speak { summon Wizard "Gandalf" } "friend";
```

## 间接对象的语法问题

这两种语法中总有一种语法更可读一些。间接对象语法不算太乱，不过有些形式存在语法二义性。首先是间接对象调用的`LIST`部分与其他列表操作符采用同样的方式解析。因此，对于下面这条语句：

```
enchant $sword ($pips + 2) * $cost;
```

会认为其中的小括号包围了方法的所有参数，而不论后面是什么。

因此它就等价于下面这条语句：

```
($sword->enchant($pips + 2)) * $cost;
```

这通常并不是你想要的（调用`enchant`时只为方法提供`$pips + 2`，然后将方法的返回值乘以`$cost`）。与其他列表操作符一样，还要当心`&&`和`||`相对于`and`和`or`的优先级（如果没有小括号）。

例如，以下代码：

```
name $sword $oldname || "Glamdring"; # 这里不能用"or"！
```

会变成以下调用，这很明确：



```
$sword->name($oldname || "Glamdring");
```

不过以下代码：

```
speak $mage "friend" && enter(); # 这里应该用"and" 才对!
```

则是不确定的：

```
$mage->speak("friend" && enter());
```

要修正这个问题，可以重写为以下等价形式：

```
enter() if $mage->speak("friend");  
$mage->speak("friend") && enter();  
speak $mage "friend" and enter();
```

间接对象形式的第二个语法问题是它的*INVOCANT*只能是一个名字、无下标的标量变量或者一个代码块<sup>注4</sup>。一旦解析器看到这样一个语法项，就有了*INVOCANT*，所以接下来会开始查找LIST。因此，以下调用：

```
move $party->{LEADER};      # 可能有错误!  
move $riders[$i];          # 可能有错误!
```

实际上会解析为：

```
$party->move->{LEADER};  
$riders->move([$i]);
```

而不是你希望的：

```
$party->{LEADER}->move;  
$riders[$i]->move;
```

解析器为间接对象查找调用者时只是稍稍向前看一点，甚至还不及查找一元操作符时看得远。如果使用第一种表示法就不会出现这个问题，所以你可能想一直选择箭头方式作为你的“武器”。

甚至英语在这方面也存在一个类似的问题。考虑这样一个命令“Throw your cat out the window a toy mouse to play with”（向窗外扔一个玩具老鼠给猫玩）。如果解析这个句子过快，可能会把猫扔出去，而不是老鼠（除非你注意到猫已经在窗外）。与Perl类似，英语中要表达这个意思有两种不同的语法：“Throw your cat the mouse”（给猫扔老鼠）和“Throw the mouse to your cat”（把老鼠扔给猫）。有时比较长的形式更清晰，也更自然，不过有时比较短的反面更好。至少，Perl中所有复杂的间接对象两边都需要加上大括号。

---

注4：细心的读者可能会记得，趣味字符后面允许出现一些语法项来指示变量解引用，那里允许的语法项与间接对象形式中*INVOCANT*所允许的语法项完全相同，例如，`@ary`，`@$aryref`或`@{$aryref}`。

## 引用包的类

关于间接对象方式的方法调用，最后一个语法二义性问题是它可能根本不会解析为一个方法调用，因为当前包可能有一个与方法同名的子例程。使用类方法时（以一个包名作为调用者），可以采用以下方法解决这个二义性，同时仍保持间接对象语法：可以在类名后面追加一个双冒号，指定这个类引用包（package-quote）。

```
$obj = method CLASS::; # 强制为"CLASS" ->method
```

这很重要，因为通常的记法是：

```
$obj = new CLASS;      # 可能并不解析为方法
```

如果当前包有一个名为new或CLASS的子例程，那么这种记法不一定总有正确的表现。即使你很注意地使用箭头形式而不是间接对象形式来调用方法，（在极少数情况下）仍有可能出问题。尽管增加了额外的标点符号，但这个代价是值得的，CLASS::记法可以保证Perl正确解析你的方法调用。下面的前两个例子解析结果不一定相同，不过，后面的两个例子总会得到相同的解析结果：

```
$obj = new ElvenRing;      # 可能是new( "ElvenRing" )
                           # 或者甚至可能是new(ElvenRing())
$obj = ElvenRing->new;      # 可能是ElvenRing()->new()

$obj = new ElvenRing::;    # 总是"ElvenRing" ->new()
$obj = ElvenRing::->new;    # 总是"ElvenRing" ->new()
```

加一点创意，将代码对齐，可以让这个包引用记法更漂亮：

```
$obj = new ElvenRing::
      name => "Narya",
      owner => "Gandalf",
      domain => "fire",
      stone => "ruby";
```

不过，看到这个双冒号你可能还是会说“噢，真难看！”所以可以告诉你，只要能保证以下两点，完全可以只使用一个裸类名。首先，没有与类同名的子例程（如果你遵循约定，即类似new的子例程名首字母小写，而类似ElvenRing的类名首字母大写，就不会有这个问题）。其次，类已经用以下的一个声明加载：

```
use ElvenRing;
require ElvenRing;
```

这些声明可以确保Perl知道ElvenRing是一个模块名，这会强制将类名ElvenRing前面的裸名（如new）解释为一个方法调用，即使你在当前包中刚好声明了你自己的一个子例程new，仍会正确地解析为方法调用。人们使用间接对象时通常不会有麻烦，除非把多个类都堆在同一个文件中，在这种情况下，Perl可能不知道某个特定的包名要作为一个类名。如果有人将子例程命名为类似ModuleNames的名字，结局也会很悲惨。

如果做以下声明，（几乎）就会遇到这种情况：

```
no feature "switch";
```

假设你使用了推荐的`use v5.14`之类的声明，v5.10或以上版本都引入了`break`，它作为一个关键字用来支持`given`构造。我们关闭了“`switch`”特性，因为如果不关闭这个特性，编译器会认为这个`break`可能是一个关键字。这里在末尾增加小括号也没有帮助，尽管你通常这么做（或者应当这么做）来保证使用这种语法的方法调用是安全的。编译器实际上并不知道要用它做什么，不过不会听之任之。

## 对象构造

所有对象都是引用，不过并不是所有引用都是对象。引用一般并不作为对象，除非它的指示对象有特殊标志告诉Perl它属于哪个包。用一个包名（即类，因为类就是一个包）标记指示对象的行为称为祝福（`blessing`）。可以认为祝福是把引用转换为一个对象的过程，不过更准确的说法是，它把引用转换为一个对象引用。

`bless`函数可以有一个或两个参数。第一个参数是一个引用，第二个参数是祝福指示对象所属的包。如果忽略第二个参数，则使用当前包。

```
$obj = { };           # 得到匿名散列的引用。
bless($obj);          # 祝福散列属于当前包。
bless($obj, "Critic"); # 祝福散列属于类Critic。
```

这里我们使用了一个匿名散列的引用，人们通常都使用散列作为对象的数据结构。毕竟散列极为灵活。不过，需要强调，你可以祝福一个引用，使它转换为Perl中能作为引用的任何东西，包括标量、数组、子例程和类型团。如果你认为有充分的理由，甚至还可以将引用祝福为一个包的符号表散列（或者，即使找不到这样的理由，也可以这样做）。Perl中的面向对象与数据结构是完全正交的。

一旦祝福指示对象，在其引用上调用内置的`ref`函数会返回所祝福的类名，而不是内置类型，如`HASH`。如果你想得到内置类型，可以使用`attributes`模块中的`reftype`函数。参见第29章中的“`attributes`”一节。

对象就是这样构造的。只需要得到某个类型的引用，将它祝福到一个包，为它指定一个类，就大功告成了。如果你要设计一个最小的类，以上就是要做的全部工作。如果你要使用一个类，甚至更简单，因为类的作者已经把这个`bless`隐藏在一个称为构造函数（`constructor`）的方法中，这个方法会创建并返回类的实例。由于`bless`返回其第一个参数，所以典型的构造函数可以非常简单，如下所示：

```
package Critter;
sub spawn { bless {} }
```



或者，也可以描述得更明确一些：

```
package Critter;
sub spawn {
    my $self = {};          # 一个空匿名散列的引用
    bless $self, "Critter"; # 使这个散列转换为一个Critter对象
    return $self;           # 返回新生成的Critter
}
```

有了这个定义，可以如下创建一个Critter对象：

```
$pet = Critter->spawn;
```

## 可继承的构造函数

像所有方法一样，构造函数也只是一个子例程，不过我们不会称它是子例程。我们总是把它作为一个方法来调用。在这个特殊情况下，这是一个类方法，因为调用者是一个包名。方法调用与常规的子例程调用有两点区别。首先，方法调用会得到一个额外的参数，这在前面已经讨论过。其次，方法调用遵循继承原则，允许一个类使用另一个类的方法。

下一节还会更严格地介绍继承的底层原理，不过现在先给出几个简单的例子来说明继承的作用，这将有助于你设计构造函数。例如，假设我们有一个Spider类，它从Critter类继承方法。具体地，假设Spider类没有自己的spawn方法。这里会应用表12-1所示的对应关系。

表12-1：方法映射到子例程

方法调用	相应的子例程调用
Critter->spawn()	Critter::spawn("Critter")
Spider->spawn()	Critter::spawn("Spider")

这两种情况下，调用的子例程是一样的，不过参数有所不同。注意上面的spawn构造函数完全忽略了参数，这说明我们的Spider对象被错误地祝福到类Critter。更好的构造函数应当为bless提供包名（作为第一个参数传入）：

```
sub spawn {
    my $class = shift;      # 存储包名
    my $self = { };
    bless($self, $class);   # 将引用祝福到这个包
    return $self;
}
```

现在可以对这两种情况使用同一个子例程：

```
$vermin = Critter->spawn;
$shelob = Spider->spawn;
```

每个对象都会有正确的类。甚至可以间接设置，如下：

```
$type = "Spider";  
$shelob = $type->spawn;      # 等同于"Spider" ->spawn
```

这仍是一个类方法，而不是实例方法，因为它的调用者包含一个字符串而不是一个引用。

如果\$type是一个对象而不是一个类名，前面的构造函数定义就不起作用了，因为bless需要一个类名。不过，对于很多类来说，完全可以使用一个现有的对象作为模板，基于这个模板来创建另一个对象。在这种情况下，可以将你的构造函数设计为既能接受对象也能接受类名：

```
sub spawn {  
    my $invocant = shift;  
    my $class    = ref($invocant) || $invocant; # 对象或类名  
    my $self     = { };  
    bless($self, $class);  
    return $self;  
}
```

## 初始化方法

大多数对象都会维护一些内部信息，这些信息由对象的方法间接管理。到目前为止，我们的所有构造函数都只是创建了空散列，不过没有理由让它们一直为空。例如，可以让构造函数接收额外的参数，将它们作为键/值对存储在这个散列中。面向对象文献通常称这些数据为属性（properties或attributes），有的文献也称为存取方法（accessors或accessor method）、成员数据（member data）、实例数据（instance data）或实例变量（instance variables）。本章后面的“实例变量”一节会更详细地讨论属性。

假设一个Horse类包含一些实例属性，如“name”和“color”：

```
$steed = Horse->new(name => "Shadowfax", color => "white");
```

如果这个对象实现为一个散列引用，一旦从参数表删除调用者，键/值对可以直接内插到散列中：

```
sub new {  
    my $invocant = shift;  
    my $class = ref($invocant) || $invocant;  
    my $self = { @_ };      # 其余的参数变成属性  
    bless($self, $class);   # 赋予对象性  
    return $self;  
}
```

这一次我们使用了一个名为new的方法作为这个类的构造函数，看到这个“new”，C++程序员可能会由此推断他们知道将要做什么。不过Perl并不认为“new”有什么特殊的含义；可以为构造函数命名为你喜欢的任何名字。只要创建并返回一个对象的方法实际上都

是构造函数。一般地，我们建议命名构造函数时，应当指定一个在当前问题上下文中有意义的名字。例如，Tk模块中的构造函数就是按它们创建的部件（widget）命名的。在DBI模块中，名为connect的构造函数会返回一个数据库句柄对象，另一个名为prepare的构造函数会作为一个实例方法调用，并返回一个语句句柄对象。不过，如果没有特定于上下文的合适的构造函数名，new可以作为一个不坏的选择。重申一句，选择一个随机的名字也不一定是坏事，这样一来，人们在使用构造函数前可能不得不仔细阅读接口契约（也就是类文档）。

如果更复杂一些，可以用默认的键/值对建立构造函数，以后用户可以提供自己的键/值对作为参数来覆盖这些默认属性：

```
sub new {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    my $self = {
        color => "bay",
        legs => 4,
        owner => undef,
        @_, # 覆盖原来的属性
    };
    return bless $self, $class;
}

$ed = Horse->new;                # 4条腿枣红马
$stallion = Horse->new(color => "black"); # 4条腿黑马
```

这个Horse构造函数用作为一个实例方法时，会忽略其调用者现有的属性。你可以另外设计一个构造函数作为实例方法来调用，如果设计合理，可以使用来自调用对象的值作为这个新构造函数的默认值：

```
$steed = Horse->new(color => "dun");
$foal = $steed->clone(owner => "EquuGen Guild, Ltd.");

sub clone {
    my $model = shift;
    my $self = $model->new(%$model, @_);
    return $self; # Previously blessed by ->new
}
```

还可以把这个功能直接放在new中，不过这个名字不太适合这个功能。

注意，甚至在clone构造函数中我们也没有硬编码Horse类的名字。我们让原对象调用自己的new方法（不论它是什么）。如果硬编码写为Horse->new而不是\$model->new，这个类就无法支持Zebra或Unicorn类的继承。你本来想克隆一匹飞马，却突然发现得到的只是一匹颜色不同的普通马，这肯定不是你想要的。

不过，有时你会遇到一个相反的问题：并不是多个类共享同一个构造函数，你可能需要多个构造函数共享一个类的对象。如果构造函数想调用一个基类的构造函数来完成部分构造



工作，就会出现这种情况。Perl不会为你完成分层构造。也就是说，Perl不会自动地调用请求类的任何基类的构造函数（或析构函数），所以你的构造函数必须自己完成这个工作（调用基类的构造函数），然后再增加派生类需要的额外属性。所以这种情况与clone子例程是类似的，只不过并不是把一个现有的对象复制到新对象，而是调用基类的构造函数，然后让这个新的基类对象“变形”为新的派生对象。

## 类继承

与Perl对象系统中的其他概念类似，要让一个类继承另一个类，并不需要在Perl中增加任何特殊的语法。调用一个方法时，如果Perl发现调用者包中没有相应的子例程，就会检查这个包的@ISA数组<sup>注5</sup>。Perl采用以下方法实现继承：给定包的@ISA数组中的各个元素分别包含另一个包的包名，如果在给定包中没有找到要调用的方法，就会在这些包中查找。例如，下面的代码会使Horse类成为Criticter类的一个子类（我们用our声明@ISA，因为这必须是一个包变量，而不能是用my声明的词法作用域变量）。

```
package Horse;
our @ISA = "Criticter";
```

可能还会看到parent pragma，它会为你处理@ISA并同时加载父类：

```
package Horse;
use parent qw(Criticter);
```

parent pragma取代了较早的base pragma，base pragma的工作是一样的，只不过如果它认为超类用到了fields魔法，也会加入一些fields魔法。如果你不知道这是什么，不用担心（用parent就可以了）：

```
package Horse;
use base qw(Criticter).
```

这样一来，原来使用Criticter的所有地方都可以使用Horse类或对象。如果你的新类通过了这个空子类测试（empty subclass test），可以认为Criticter是一个适合继承的合适的基类。

假设有一个Horse对象存储在\$steed中，并在这个对象上调用一个move方法：

```
$steed->move(10);
```

因为\$steed是一个Horse，对于这个方法，Perl的第一个选择就是Horse::move子例程。如果没有这样一个子例程，并不是产生一个运行时异常，Perl会查看@Horse::ISA的第一个元素，这会指示它在Criticter包中查找Criticter::move。如果这个子例程也没有找到，而且Criticter有它自己的@Criticter::ISA数组，会继续在这个数组中查找包名，希望找到一个可

---

注5： 读作“is a”（是一个），如“A horse is a critter”（马是一种家畜）。

能提供`move`方法的祖先包，依此类推，按照继承层次回溯，直到到达一个没有`@ISA`的包。

我们介绍的这种情况是单一继承（single inheritance），即每个类只有一个父类。这种继承类似于一个相关包的链表。Perl还支持多重继承（multiple inheritance）；只需要向类的`@ISA`增加多个包就可以实现多重继承。这种继承更类似于一个树数据结构，因为每个包可以有多个直接父类。有些人认为这样更诱人。

在一个类型为`classname`的调用者上调用一个名为`methname`的方法时，Perl会尝试6种不同方法来找到要使用的子例程：

1. 首先，Perl在调用者自己的包中查找一个名为`classname::methname`的子例程。如果失败，则继承介入，进入第2步。
2. 接下来，Perl检查从基类继承的方法，在`@classname::ISA`中所列的所有`parent`包中查找`parent::methname`子例程。这里会从左向右采用深度优先方式递归搜索。递归可以确保祖父类、曾祖父类、曾曾祖父类等都会搜索到。
3. 如果失败，Perl查找一个名为`UNIVERSAL::methname`的子例程。
4. 如果进入这一步，Perl放弃查找`methname`，开始查找一个`AUTOLOAD`。首先，它查找一个名为`classname::AUTOLOAD`的子例程。
5. 如果失败，Perl会搜索`@classname::ISA`中所列的所有`parent`包，查找`parent::AUTOLOAD`子例程。这个搜索同样是从左向右采用深度优先方式的递归搜索。
6. 最后，Perl查找一个名为`UNIVERSAL::AUTOLOAD`的子例程。

成功找到第一个子例程时，Perl停止搜索，并调用这个子例程。如果没有找到子例程，会产生一个异常，这是我们经常看到的一个异常：

```
Can't locate object method "methname" via package "classname"
```

如果对C编译器使用`-DDEBUGGING`选项，可以构建一个调试版本的Perl，通过使用Perl的`-Do`开关，可以查看完成方法调用解析时所经历的每一个步骤。

在后面的介绍中我们还会更详细地讨论继承机制。

## 通过@ISA继承

如果`@ISA`包含不只一个包名，默认地会按从左到右的顺序搜索所有这些包。这是一个深度优先的搜索，所以如果有一个`Mule`类，建立了以下继承关系：

```
package Mule;
our @ISA = ("Horse", "Donkey");
```



对于Mule中没有的方法，Perl首先会在Horse（以及它的所有祖先类，如Criticter）中查找，然后才会继续搜索Donkey及其祖先类。

如果在某个基类中找到了当前类中缺少的方法，Perl会把这个位置缓存在当前类内部，以提高效率，这样下一次需要查找这个方法时就不用再那么兴师动众了。一旦改变了@ISA或者定义了新方法，都会使这个缓存失效，需要Perl重新查找。

Perl搜索一个方法时，它会确保你没有创建一个循环的继承体系。如果两个类相互继承就会出现这种情况，甚至通过其他类间接继承也可能出现循环继承。就像你要做自己的曾祖父一样，即使对于Perl这也太荒谬了，所以这种尝试会产生一个异常。不过，如果继承多个类，而它们都有一个共同的祖先，在Perl看来这不是错误，这有点像近亲结婚。这种继承体系不再像一个树，而更像一个有向无环图。这不会让Perl为难，只要这个图确实是无环的。

设置@ISA时，赋值通常在运行时发生，所以除非你特别当心，否则BEGIN、CHECK、UNITCHECK或INIT块中的代码不能使用这个继承体系。一个防范措施是使用parent pragma，这允许你使用require加载类，并在编译时把它们增加到@ISA。可以如下使用：

```
package Mule;
use parent ("Horse", "Donkey"); # 声明超类
```

它是以下代码的简写形式：

```
package Mule;
BEGIN {
    our @ISA = ("Horse", "Donkey");
    require Horse;
    require Donkey;
}
```

有时让人们奇怪的是，在@ISA包含一个类并不会用require加载适当的模块。这是因为，Perl的类系统与其模块系统很大程度上是正交的。一个文件可以包含多个类（因为它们都是包），也可以在多个文件中提到一个包。不过，在最常见的情况下，如果你不那么苛刻，包、类、模块和文件完全可以互换，parent pragma提供了一种声明性语法，可以建立继承并加载模块文件。这也是我们一直在谈到的便利正交性之一。

关于use parent的更多详细内容请参见第29章的介绍。另外可以参见较早的base pragma，它能提供额外的fields魔法（现在这个特性已经被Perl程序员冷落）。

## 候选方法搜索

对于多重继承，默认的遍历@INC来找到合适方法的做法可能就不适用了，因为一个较远的超类中的方法可能会隐藏较近的超类中的方法，尽管较近的超类中的方法更好。考虑图



12-1所示的继承关系，这里Mule继承了两个类：Donkey和Horse，它们都继承自Equine。Equine有一个color方法，Donkey继承了这个方法。不过Horse提供了它自己的color。使用默认遍历，除非你知道父类的顺序，否则就无法知道你会得到哪一个color：

```
use parent qw(Horse Donkey); # 先找到Horse::Color
use parent qw(Donkey Horse); # 先找到Equine::Color
```

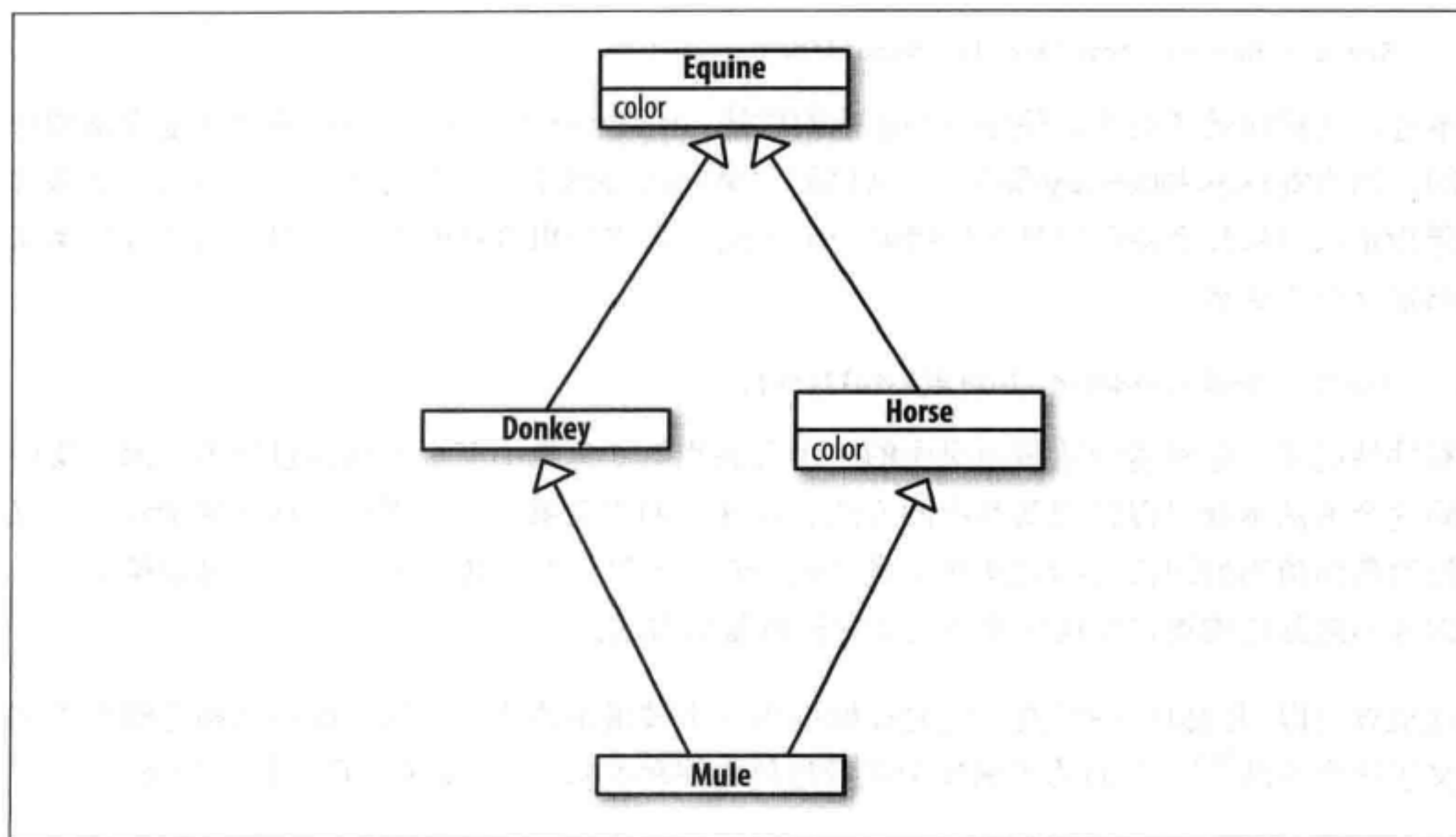


图12-1：多重继承图

在v5.10中，这个遍历是可配置的。用专业术语来讲，这就是方法解析顺序（method resolution order），可以用mro pragma（见第29章）选择：

```
package Mule;
use mro 'c3';
use parent qw(Donkey Horse);
```

C3算法会遍历@INC，从而找到继承图中更近的继承方法。换句话说，这表示只有当搜索了超类的所有子类之后才会搜索该超类。Perl查看Horse之前不会查看Equine。

如果你的Perl不支持mro pragma，可以使用MRO::Compat CPAN模块。

## 访问被覆盖的方法

类定义一个方法时，这个子例程会覆盖所有基类中的同名方法。如果有一个Mule对象（派生自类Horse和类Donkey），你想调用这个对象的breed方法。尽管父类都有自己的breed方法，但Mule类的设计者覆盖了这些方法，为Mule类提供了它自己的breed方法。这意味着以下杂交不太可能有结果：

```
$stallion = Horse->new(gender => "male");
$molly = Mule->new(gender => "female");
$colt = $molly->breed($stallion);
```

现在，假设基因工程创造了奇迹，你找到一种方法可以解决骡子不能生育的问题，所以你想跳过这个不能生育的Mule::breed方法。可以作为一个普通的子例程来调用方法，但要确保显式地传入调用者：

```
$colt = Horse::breed($molly, $stallion);
```

不过，这样回避了继承，往往不是正确的做法。Horse::breed子例程不存在是无法想象的，因为Horses和Donkey都从一个共同的父类Equine派生了这个行为。另一方面，如果你想指定Perl从某个特定的类开始搜索一个方法，只需使用普通的方法调用，只是要用类来限定这个方法名：

```
$colt = $molly->Horse::breed($stallion);
```

有些情况下，你可能希望派生类中的一个方法相当于基类中某个方法的包装器。派生类中的这个方法本身可以调用基类中的方法，在这个调用之前或之后增加它自己的动作。可以使用前面给出的记法来指定从哪个类开始搜索。不过，对于覆盖的方法，大多数情况下，你并不想知道或指定要执行哪个父类中被覆盖的方法。

这里就可以用到SUPER伪类。它允许你调用一个被覆盖的基类方法，而不必指定哪个类定义了这个方法<sup>注6</sup>。下面的子例程会查看当前包的@ISA，而不要求你指定特定的类：

```
package Mule;
our @ISA = qw(Horse Donkey);
sub kick {
    my $self = shift;
    print "The mule kicks!\n";
    $self->SUPER::kick(@_);
}
```

只有在方法内部使用SUPER伪包才有意义。实现一个类时，有人可能会在代码中使用SUPER，这是可以的，不过如果只是要使用一个类的对象，就不能使用SUPER。

如果使用C3方法解析顺序，那么并不是使用SUPER::METHNAME，而应当使用next::method，这是由use mro "c3" pragma加载的。与SUPER不同，利用next::method，你不用指定方法名，因为next::method会为你确定方法：

```
use v5.14;
package Mule;
use mro 'c3';
use parent qw(Horse Donkey);
```

---

注6：不要把它与第11章提到的覆盖Perl内置函数的机制混淆了，内置函数是对象方法，所以不会因继承而被覆盖。可以通过CORE伪包而不是SUPER伪包来覆盖内置函数。

```

sub kick {
    my $self = shift;
    say "The mule kicks!";
    $self->next::method(@_);
}

```

Perl中的所有代码都知道当前包是什么，这由最后一个package语句确定。编译SUPER调用时，SUPER方法只会在当前包的@ISA中查找。它不关心调用者的类，也不关心被调用的子例程所属的包。如果你想在另一个类中定义方法，而且只改变了方法名，这可能会有问题：

```

package Bird;
use Dragonfly;
sub Dragonfly::divebomb { shift->SUPER::divebomb(@_) }

```

很遗憾，这会调用Bird的超类，而不是Dragonfly的超类。要达到我们真正的目的，你还需要显式地切换到适当的包来完成SUPER编译：

```

package Bird;
use Dragonfly;
{
    package Dragonfly;
    sub divebomb { shift->SUPER::divebomb(@_) }
}

```

next::method也存在一个类似的问题，因为它使用了调用者（caller）的包来确定要查看哪个类。如果你在另一个包中为Donkey定义一个方法，next::method就会出问题：

```

package main;
*Donkey::sound = sub { (shift)->next::method(@_) };

```

这个匿名子例程在堆栈中作为\_\_ANON\_\_出现，所以next::method并不知道它在哪个包中。不过，可以使用Sub::Name CPAN模块来确定它所在的包：

```

use Sub::Name qw(subname);
*Donkey::sound =
    subname 'Donkey::sound' => sub { (shift)->next::method(@_) };

```

如这些例子所示，如果只是要向一个现有的类增加方法，并不需要为此编辑模块文件。因为类就是包，方法就是子例程，就像我们一样，你要做的只是在包中定义一个函数，这样类就会立即有了一个新方法。根本不需要任何继承。重要的是包，因为包是全局的，可以在程序中的任何位置访问所有包（我们提到过吗？我们打算下星期在你的卧室安装一个按摩浴缸）。

## UNIVERSAL: 终极祖先类

递归地搜索调用者的类以及其所有祖先类之后，如果没有找到指定的方法定义，要找到这个方法还可以做一个检查：可以在一个特殊的预定义类UNIVERSAL中查找。这个包绝对



不会出现在@ISA中，不过如果@ISA检查失败，总会查看这个包。可以把UNIVERSAL看作是终极祖先类，所有类都隐含地派生自这个类，这就类似于Java中的Object类或Python中的object类。

下面是UNIVERSAL提供的预定义方法，因此所有类都包含这些方法。不论作为类方法还是作为对象方法来调用都是可以的。

#### INVOCANT->isa(CLASS)

如果INVOCANT的类是CLASS或者从CLASS继承的某个类，则isa方法返回true。除了包名以外，CLASS还可以是某个内置类型，如“HASH”或“ARRAY”（不过，像这样检查具体的类型对于封装或多态来说并不是好事。应当依靠方法分派来提供正确的方法）。

```
use IO::Handle;
if (IO::Handle->isa("Exporter")) {
    print "IO::Handle is an Exporter.\n";
}

$fh = IO::Handle->new();
if ($fh->isa("IO::Handle")) {
    print "\$fh is some sort of IOish object.\n";
}
if ($fh->isa("GLOB")) {
    print "\$fh is really a GLOB reference.\n";
}
```

#### INVOCANT->DOES(ROLE)

Perl v5.10增加了角色（roles）的概念，采用这种方式，类可以包含外部方法，而不用像isa一样要求必须继承那些方法。角色指定了一组行为，但是并不关心类如何实现这些行为。它可能继承这些方法，也可能模仿、委托或做其他处理。

默认的，DOES等同于isa，所有情况下都可以使用DOES取代isa。不过，如果你的类做了一些有趣的处理，可以包含方法而没有使用继承，在这种情况下，你可能希望定义DOES来给出正确的答案。

角色是Perl 6新增的内容，事实上Perl 5对于角色没有做任何处理。UNIVERSAL DOES方法的存在是为了让合作的类（采用这种方式包含的类）构建一个环境，在这个环境下能体现出DOES重要性。对于这一点Perl本身并不关心。

#### INVOCANT->can(METHOD)

如果对INVOCANT应用了METHOD，can方法会返回将要调用的子例程的一个引用。如果没有找到这样一个子例程，can会返回undef。

```
if ($invocant->can("copy")) {
    print "Our invocant can copy.\n";
}
```

可以利用`can`有条件地调用一个方法，仅当方法存在时才调用：

```
$obj->snarl if $obj->can("snarl");
```

在多重继承情况下，这允许一个方法调用所有被覆盖的基类的方法，而不只是最左基类的方法：

```
sub snarl {
    my $self = shift;
    print "Snarling: @_\\n";
    my %seen;
    for my $parent (@ISA) {
        if (my $code = $parent->can("snarl")) {
            $self->$code(@_) unless $seen{$code}++;
        }
    }
}
```

我们使用`%seen`散列来记录已经调用了哪些子例程，这样可以避免多次调用同一个子例程。如果多个父类有一个共同的祖先，就可能出现这种情况。

触发`AUTOLOAD`（下一节将介绍）的方法不会准确地报告，除非包已经声明了（但未定义）它希望自动加载的子例程。

如果使用`mro pragma`，可能更愿意使用`next::can`方法而不是这个方法。

#### `INVOCANT->VERSION(NEED)`

`VERSION`方法返回`INVOCANT`的类的版本号，这个版本号存储在包的`$VERSION`变量中。

如果提供了`NEED`参数，它会验证当前版本不小于`NEED`，如果小于`NEED`则产生一个异常。`use`确定一个模块是否足够新时就是调用这个方法实现的。

```
use Thread 1.0; # 调用Thread->VERSION(1.0)
print "Running version ", Thread->VERSION, " of Thread.\\n";
```

可以提供你自己的`VERSION`方法来覆盖`UNIVERSAL`的这个方法。不过，这样一来，由你的类派生的所有子类都会使用这个被覆盖的方法。如果你不希望这样，就应该适当地设计你的`VERSION`方法，把其他类的`version`请求重新委托给`UNIVERSAL`。

`UNIVERSAL`中的方法是内置Perl子例程，如果指定这些方法的完全限定名，并传递两个参数，如`UNIVERSAL::isa($formobj, "HASH")`，就可以调用这些方法。不过，这样会绕过一些安全性检查，因为`$formobj`可以是任何值，而不只是引用。可以把它包围在`eval`中：

```
eval { UNIVERSAL::isa($formobj, "HASH") }
```

不过，不推荐这种做法，因为`can`通常就有你要找的答案：

```
eval { UNIVERSAL::can($formobj, $method) }
```

如果你担心`$formobj`是一个对象，希望把它包围在一个`eval`中，也可以把它用作为一个对象，因为答案是一样的（不能在`$formobj`上调用这个方法）：

```
eval { $formobj->can( $method ) }
```

完全可以为类`UNIVERSAL`增加你自己的方法（当然，要当心，你可能会让一些人很困扰，他们本来不想找到你定义的方法名，这样就能从别处自动加载他们真正需要的方法）。这里我们创建了一个`copy`方法，所有类的对象都可以使用这个方法（如果它们没有定义自己的`copy`方法）。如果在类上调用这个方法而不是在对象上调用，则会失败：

```
use Data::Dumper;
use Carp;
sub UNIVERSAL::copy {
    my $self = shift;
    if (ref $self) {
        return eval Dumper($self); # no CODE refs
    } else {
        confess "UNIVERSAL::copy can't copy class $self";
    }
}
```

如果对象包含子例程的引用，这个`Data::Dumper`策略就不奏效了，因为这些引用不能正确的再生。即使引用源可用，但词法作用域绑定会丢失。

## 方法自动加载

正常情况下，调用包中一个未定义的子例程时，如果这个包定义了一个`AUTOLOAD`子例程，那么会调用这个`AUTOLOAD`子例程而不是产生一个异常（参见第10章中“自动加载”一节）。方法的处理稍有不同。如果常规的方法搜索（搜索类、其祖先类，以及最后的`UNIVERSAL`）未能找到一个匹配，就会再按同样的序列进行一次搜索，不过这一次查找的是`AUTOLOAD`子例程。如果找到，就会将这个子例程作为方法来调用，将包的`$AUTOLOAD`变量设置为这个子例程的完全限定名（即`AUTOLOAD`所代表的要调用的子例程）。

自动加载方法时需要更加小心。首先，如果`AUTOLOAD`子例程代表一个名为`DESTROY`的方法调用，则应立即返回，除非你的目标就是模拟`DESTROY`，它对Perl有特殊的含义（参见本章后面的“实例析构函数”一节）。

```
sub AUTOLOAD {
    return if our $AUTOLOAD =~ /::DESTROY$/;
    ...
}
```

其次，如果类提供了一个`AUTOLOAD`安全网，你不能在一个方法名上使用`UNIVERSAL::can`来检查是否能安全地调用。必须单独地检查`AUTOLOAD`：

```
if ($obj->can("methname") || $obj->can("AUTOLOAD")) {
```



```
    $obj->methname();  
}
```

最后，在多重继承情况下，如果一个类继承了两个或多个类，而且每个父类都有一个AUTOLOAD，那么只会触发最左父类，因为Perl一旦找到第一个AUTOLOAD就会停止搜索。

后面两个问题很容易避开，只需要在适当的包中声明子例程，要由这个包的AUTOLOAD管理这些方法。可以分别声明：

```
package Goblin;  
sub kick;  
sub bite;  
sub scratch;
```

或者利用subs pragma来声明，如果需要声明多个方法，这种做法更方便：

```
package Goblin;  
use subs qw(kick bite scratch);
```

即使只是声明了这些子例程，而没有定义，这也足以让系统认为它们是真实。它们会出现在UNIVERSAL::can检查中，更重要的是，它们会出现在方法搜索的第2步，这样就不会进入第3步，更不会进入第4步。

“不过，”你可能会辩解，“它们调用了AUTOLOAD，不是吗？”嗯，没错，最后确实调用了AUTOLOAD，不过原理是不同的。通过第2步找到方法桩（stub）之后，Perl尝试调用这个方法。当它发现这个方法根本不是它想要的方法时，就会再次开始AUTOLOAD方法搜索。不过，这一次，它首先从包含这个桩的类开始搜索，这就将方法搜索限制在这个类及其祖先类（以及UNIVERSAL）中。采用这种方式，Perl会找到要运行的正确的AUTOLOAD，并且知道要忽略原来继承树中不合适的AUTOLOAD。

## 私有方法

还有一种调用方法的方式，采用这种方式时，Perl可以完全忽略继承。如果不是使用直接量方法名，而是指定了一个简单的标准变量，其中包含一个子例程的引用，就会立即调用这个子例程。在上一节UNIVERSAL->can的介绍中，最后一个例子就是使用子例程的引用来调用所有覆盖的方法，而不是使用方法名。

这种行为有一个很诱人的方面，可以用来实现私有方法调用。如果把你的类放在一个模块中，可以利用文件的词法作用域来实现私有性。首先，将一个匿名子例程保存在一个文件作用域的词法作用域变量中：

```
# 声明私有方法  
my $secret_door = sub {  
    my $self = shift;  
    ...  
}
```

```
};
```

然后可以在文件中使用这个变量，就好像它包含一个方法名一样。可以直接调用闭包，而不用考虑继承。与其他方法一样，需要传入调用者作为额外的参数。

```
sub knock {
    my $self = shift;
    if ($self->{knocked}++ > 5) {
        $self->$secret_door();
    }
}
```

这会启用文件自己的子例程（类方法）来调用方法，而这个词法作用域之外的代码不能访问这个方法。

## 实例析构函数

与Perl中所有其他指示对象一样，对象的最后一个引用删除后，会隐式地回收对象的内存。即将回收一个对象时，你还有机会加以控制，为此可以在类的包中定义一个DESTROY子例程。这个方法会在适当的时候自动触发，将回收的对象是它的唯一参数。

析构函数在Perl中很少用到，因为Perl会为你自动处理内存管理。不过，有些对象可能在内存系统之外还留有需要处理的状态，如文件句柄或数据库连接。

```
package MailNotify;
sub DESTROY {
    my $self = shift;
    my $fh = $self->{mailhandle};
    my $id = $self->{name};
    print $fh "\n$id is signing off at " . localtime() . "\n";
    close $fh; # 关闭邮件程序管道
}
```

Perl只使用一个方法来构造对象，即使包含这个构造函数的类继承了一个或多个其他类，类似的，Perl对于每一个要撤销的对象也使用一个DESTROY方法，而不考虑继承。换句话说，Perl不会按继承层次为你撤销对象。如果你的类覆盖了一个超类的析构函数，那么你的DESTROY方法可能需要调用所有相关基类的DESTROY方法：

```
sub DESTROY {
    my $self = shift;
    # 检查覆盖的析构函数...
    $self->SUPER::DESTROY if $self->can("SUPER::DESTROY");
    # 现在完成你自己的处理
}
```

这只适用于继承的类。如果一个对象只是包含在当前对象中（例如，包含在更大散列中的值），它会自动释放和撤销。正是因为这个原因，通过简单的聚合建立的包含关系（有时



称为“has-a”关系)通常要比继承(即“is-a”关系)更简洁、更清晰。换句话说,实际上通常你需要把一个对象直接存储在另一个对象中,而不是通过继承,继承会增加不必要的复杂性。有时用户想使用多重继承,而实际上单一继承就足够了。

显式地调用DESTROY是可以的,不过很少需要这样调用。这甚至可能是有害的,因为在同一个对象上运行多次析构函数会有让人不快的后果。

## 利用DESTROY方法进行垃圾回收

第8章“垃圾回收、循环引用和弱引用”一节中介绍过,指示自身的变量(或者多个变量相互间接引用)不会释放,直到程序(或嵌入式解释器)快要退出时才会释放。如果你想更早地回收这些内存,通常需要显式中断引用,或者使用Scalar::Util模块将其弱化。

对于对象,一种候选方案是创建一个容器类,其中包含的指针指向这个自引用的数据结构。为包含对象的类定义一个DESTROY方法,手动地中断自引用结构中的循环。可以在《Perl Cookbook》的技巧13.13“Coping with Circular Data Structures Using Weak References”(使用弱引用处理循环数据结构)中找到这样一个例子。

解释器关闭时,所有对象都会撤销,这对于多线程或嵌入式Perl应用很重要。对象总是先于常规引用在前一轮撤销。这样是为了防止DESTROY方法使用那已经撤销的引用(另外也因为常规引用只在嵌入式解释器中垃圾回收,因为退出进程是回收引用的一种便捷方式。不过退出并不会运行对象析构函数,所以Perl会先调用析构函数)。

## 管理实例数据

大多数类创建的对象实际上就是一些数据结构,其中包含多个内部数据字段(实例变量),以及管理这些数据字段的方法。

Perl类只能继承方法,而不能继承数据,不过只要对对象的所有访问都通过方法调用来完成,这就没有任何问题。如果你希望做到数据继承,必须通过方法继承来达到目的。不过,这在Perl中没有必要,因为大多数类都会将其对象的属性存储在一个匿名散列中。对象的实例数据包含在这个散列中,这就相当于它自己的小命名空间,由处理对象的类来划分。例如,如果你希望一个名为\$city的对象包含一个名为elevation的数据字段,可以直接访问\$city->{elevation}。不需要任何声明。不过方法包装器还是有用的。

假设要实现一个Person对象。你打算包含一个名为“name”的数据字段,出于巧合,你要对应键name把它存储在一个匿名散列中(这个散列将作为对象)。不过你不希望用户直接接触数据。为了得到封装的好处,用户要使用方法来访问这个实例变量,而不能揭开抽象的面纱。

例如,你可能建立了一对存取方法:



```

sub get_name {
    my $self = shift;
    return $self->{name};
}

sub set_name {
    my $self = shift;
    $self->{name} = shift;
}

```

所以可以有类似下面的代码：

```

$him = Person->new();
$him->set_name("Frodo");
$him->set_name( ucfirst($him->get_name) );

```

甚至可以把两个方法合并为一个：

```

sub name {
    my $self = shift;
    if (@_) { $self->{name} = shift }
    return $self->{name};
}

```

这样可以得到类似下面的代码：

```

$him = Person->new();
$him->name("Frodo");
$him->name( ucfirst($him->name) );

```

为每个实例变量分别编写一个单独的函数有很多好处（我们的Person类可能有name、age、height等实例变量），这样很直接、明显，而且很灵活。缺点是，每次需要一个新类时，都必须为每个实例变量定义一个或两个几乎完全相同的方法。对于前面几个类，这还不算太坏，如果你愿意当然可以这么做。不过，如果方便性比灵活性更重要，你可能更愿意使用下面几节介绍的技术。

需要说明，我们改变的是实现，而不是接口。如果类的用户很看重封装，你可以透明地将一个实现切换为另一个实现，而用户不会注意到（如果继承树中的家族成员使用你的类作为子类或超类，可能不会这么宽容，因为他们比陌生人更了解你）。如果你的用户一直在偷窥类的私有事务，那么导致的不可避免的灾难就是他们自己的错误，而与你无关。要从你这一边恪守合约，你能做到的就是维持接口不变。如果试图阻止世界上所有人做那些可怕的事情，这不仅会耗费你的所有时间和精力，而且最终仍会发现你的努力都只是徒劳无功。

处理家族成员更有挑战性。如果一个子类覆盖了一个超类的属性存取方法，它该访问散列中的同一个字段吗？有人选择是，也有人选择不是，这取决于属性的性质。从通常的安全性角度出发，每个存取方法都可以在散列字段名前面加上它自己的类名作为前缀，这样子

类和超类都可以有自己的版本。下面给出几个例子，包括标准Struct::Class模块，它们都使用了这个子类安全策略。可以看到存取方法如下：

```
sub name {
    my $self = shift;
    my $field = __PACKAGE__ . "::name";
    if (@_) { $self->{$field} = shift }
    return $self->{$field};
}
```

在以下各个例子中，我们创建了一个简单的Person类，其中包括字段name、race和aliases，它们分别有相同的接口但实现完全不同。我们不会告诉你我们最喜欢哪一个，因为取决于具体的场合，所有实现我们都喜欢。每个人的品位不同。萝卜青菜各有所爱。

## 利用自动加载生成存取方法

前面我们提到过，调用一个不存在的方法时，Perl有两种不同的方式来查找AUTOLOAD方法，这取决于是否已经声明一个桩方法。可以利用这个特性来访问对象的实例数据，而无需为每一个实例编写单独的函数。在AUTOLOAD子例程中，具体调用的方法名可以从\$AUTOLOAD变量获取。考虑以下代码：

```
use Person;
$him = Person->new;
$him->name("Aragorn");
$him->race("Man");
$him->aliases( ["Strider", "Estel", "Elessar"] );
printf "%s is of the race of %s.\n", $him->name, $him->race;
print "His aliases are: ", join(", ", @{$him->aliases}), ".\n";
```

与前面一样，这个版本的Person类实现了一个包含3个字段（name、race和aliases）的数据结构：

```
package Person;
use Carp;

my %Fields = (
    "Person::name" => "unnamed",
    "Person::race" => "unknown",
    "Person::aliases" => [],
);

# 下一个声明保证我们得到自己的自动加载器
use subs qw(name race aliases);

sub new {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    my $self = { %Fields, @_ }; # 类似Class::Struct克隆
    bless $self, $class;
    return $self;
}
```

```

}

sub AUTOLOAD {
    my $self = shift;
    # 只处理实例方法，而不处理类方法
    croak "$self not an object" unless ref($invocant);
    my $name = our $AUTOLOAD;
    return if $name =~ /::DESTROY$/;
    unless (exists $self->{$name}) {
        croak "Can't access '$name' field in $self";
    }
    if (@_) { return $self->{$name} = shift }
    else { return $self->{$name} }
}

```

可以看到，这里找不到名为name、race或aliases的方法。AUTOLOAD子例程会全权负责。有人使用\$him->name(“Aragorn”)时，就会调用AUTOLOAD子例程，并将\$AUTOLOAD设置为“Person::name”。为了方便，我们使用了完全限定名，访问对象散列字段就应当采用这种形式。这样一来，如果这个类作为一个更大层次体系的一部分，就不会与其他类中使用的同名字段冲突。

## 用闭包生成存取方法

大多数存取方法实际上都在做同样的事情：就是要获取或存储这个实例变量的值。在Perl中，要创建一组几乎重复的函数，最自然的方法是循环处理一个闭包。不过，闭包是没有名字的匿名函数，而方法必须是类的包符号表中的命名子例程，这样才能根据它们的名字来调用。这没有问题，只需要把闭包引用赋给一个有适当名字的类型团。

```

package Person;

sub new {
    my $invocant = shift;
    my $self = bless({}, ref $invocant || $invocant);
    $self->init();
    return $self;
}

sub init {
    my $self = shift;
    $self->name("unnamed");
    $self->race("unknown");
    $self->aliases([]);
}

for my $field (qw(name race aliases)) {
    my $slot = "__PACKAGE__::$field";
    no strict "refs"; # 这样就可以使用类型团的符号引用
    *$slot = sub {
        my $self = shift;
        $self->{$field} = shift if @_;
        return $self->{$field};
    };
}

```



```
}
```

闭包是为实例数据创建大量存取方法的最简洁的方法。不论是对计算机来说还是对你来说这都是最高效的。不仅所有存取方法会共享相同的代码（它们只需要自己的便签簿 pads），而且以后如果你想增加另外一个属性，只需要做最小的修改：为for循环的列表再增加一个词，另外可能还要在init方法中加几行代码。

## 使用闭包实现私有对象

到目前为止，这些管理实例数据的技术都没有提供任何机制来“防范”外部访问。类以外的所有人都能打开对象的黑盒子，向里面窥视，只要他不在乎保修失效。强制私有性往往会影响人们正常完成工作。Perl的哲学是最好把数据封装起来并加上标签，比如：

```
IN CASE OF FIRE (如遇火灾)
BREAK GLASS (请打碎玻璃)
```

要尽可能遵循这个封装原则，但在紧急情况下（如需要进行调试），也允许很容易地访问内容。

不过，如果你确实想强调私有性，Perl也不会阻拦你。Perl提供了一些底层的构造模块，可以用这些构造模块在你的类及其对象外面加装一个无法逾越的私人护栏。实际上，这比很多流行面向对象语言中的保护更强。这里的关键组件是词法作用域以及其中的词法作用域变量，闭包则扮演了一个中枢角色。

在前面的“私有方法”一节，我们看到类可以使用闭包来实现在模块文件之外不可见的方法。后面我们将查看一些管理类数据的存取方法，这是一些极具私密性的数据，甚至这个类的其余部分都不能无限制地访问。这些仍是闭包的传统用法。真正有意思的方法是使用闭包作为对象自身。对象的实例变量锁在一个作用域中，只有对象自己（也就是闭包）可以自由访问这个作用域。这是一种很强的封装形式，不仅可以防止外部篡改，甚至同一个类的其他方法也必须使用适当的存取方法才能得到对象的实例数据。

下面的例子展示了这是如何实现的。对象本身以及生成的存取方法都使用闭包实现：

```
package Person;
sub new {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    my $data = {
        NAME => "unnamed",
        RACE => "unknown",
        ALIASES => [],
    };
    my $self = sub {
        my $field = shift;
```

```

#####
### 在这里完成访问检查    ###
#####
if (@_) { $data->{$field} = shift }
return $data->{$field};
};
bless($self, $class);
return $self;
}
# 生成方法名
for my $field (qw(name race aliases)) {
    no strict "refs"; # 用于访问符号表
    *$field = sub {
        my $self = shift;
        return $self->(uc $field, @_);
    };
}

```

`new`方法创建和返回的对象不再是一个散列，这与我们见过的其他构造函数不同。它是一个闭包，只能通过这个闭包来访问存储在散列中（由`$data`指示）的属性数据。一旦构造函数调用结束，就只能通过闭包来访问`$data`（即属性）。

在类似`$him->name("Bombadil")`的调用中，存储在`$self`中的调用对象就是构造函数祝福并返回的闭包。对闭包而言，除了调用它之外，没有太多工作，所以我们只是用`$self->(uc$field, @_)`来调用闭包。不要被这个箭头迷惑了。这只是一个常规的间接函数调用，而不是一个方法调用。初始参数是字符串“name”，其他的参数则是传入的所有参数<sup>注7</sup>。一旦在闭包中执行，就能再次访问`$data`中的散列引用。采用这种方式，闭包可以根据需要允许或拒绝对数据的访问。

除了闭包对象以外，任何人都不能未经允许地访问这些非常私有的实例数据，甚至包括这个类中的其他方法。它们可以尝试像`for`循环生成的方法那样调用闭包，可能就是要设置类从来没有听说过的一个实例变量。不过，很容易阻止这种做法，在构造函数中可以看到一些关于访问检查的注释，只需要在这些位置插入相应的代码。首先，插入一个通用的“导语”：

```

use Carp;
local $Carp::CarpLevel = 1; # 保证消息简短
my ($cpack, $cfile) = caller();

```

现在为各个检查插入代码。第一个检查要确保指定的属性名存在：

```

croak "No valid field '$field' in object"
    unless exists $data->{$field};

```

下面这个代码只允许来自同一个文件的调用者访问：

```

carp "Unmediated access denied to foreign file"
    unless $cfile eq __FILE__;

```

注7：当然，这种双函数调用速度很慢，不过如果你想更快一些，是不是首先应该使用对象呢？

下面这个代码只允许来自同一个包的调用者访问：

```
carp "Unmediated access denied to foreign package ${cpack}::"
unless $cpack eq __PACKAGE__;
```

以下代码的作用是：只有当调用者的类继承了我们的类，才允许它访问：

```
carp "Unmediated access denied to unfriendly class ${cpack}::"
unless $cpack->isa(__PACKAGE__);
```

所有这些检查只会阻止那些未经允许的访问。如果类用户礼貌地使用类的指定方法，就不会受到这个限制。Perl为你提供了一些工具，允许你非常挑剔。幸运的是，并不是所有人都那么挑剔。

不过有些人必须是这样。如果你在编写飞行控制软件，要求严格是一件好事。如果你希望或必须是这样的人，而且你更喜欢使用成熟的代码而不是自己完全从头开发，可以参考CPAN上已经提供的Damian Conway的Tie::SecureHash模块。这个模块实现了受限散列，支持公共、保护和私有约束。它还可以用来处理前一个例子中我们忽略的继承问题。Damian还写了一个更强大的模块Class::Contract，它在Perl灵活的对象系统之上增加了一个正式的软件工程体系。这个模块的特性表看起来就像是一个计算机科学教授讲授的软件工程课本上的一个清单<sup>注8</sup>，包括强制封装、静态继承，以及面向对象Perl的契约式设计条件检查，另外还为对象和类层次的属性、方法、构造函数和析构函数定义以及前置条件、后置条件和类不变式提供了声明语法。太棒了！

## 新技巧

在v5.6中，还可以声明一个方法来指示它返回一个左值。这是利用lvalue子例程属性实现的（不要与对象属性混淆）。利用这个实验性的特性，允许你把方法当作可以出现在等号左边的项：

```
package Critter;

sub new {
    my $class = shift;
    my $self = { pups => 0, @_ };    # 覆盖默认值
    bless $self, $class;
}

sub pups : lvalue {                # 后面将赋至pups()
    my $self = shift;
    a$self->{pups};
}

package main;
```

---

注8： 应该能猜出Damian是干什么的了。顺便说一句，我们强烈推荐他的作品《Object Oriented Perl》(Manning出版)。



```

$varmint = Critter->new(pups => 4);
$varmint->pups *= 2;                # 赋至$varmint->pups
$varmint->pups =~ s/(.)/$1$1/;      # 原地修改$varmint->pups
print $varmint->pups;                # 现在得到88个小狗

```

这样一来，你可以认为`$varmint->pups`是一个变量，同时仍遵循封装原则。参见第7章中“`lvalue`属性”一节。

如果运行一个线程版本的Perl，想确保只有一个线程可以调用某个对象的一个特定方法，可以使用`locked`和`method`属性来做到：

```

sub pups : locked method {
    ...
}

```

线程调用一个对象的`pups`方法时，Perl在执行前会锁定这个对象，防止其他线程调用同样的方法。参见第7章中“`method`属性”一节。

## 管理类数据

我们已经介绍过，可以使用多种方法来访问各个对象的数据值。不过，有时你可能还想访问一个类所有对象共有的某个共同状态。这些变量并不是类的某一个实例的属性，它们是整个类的全局变量，不论你使用哪一个类实例（对象）来访问都是一样的（C++程序员可能认为这些变量是静态成员数据）。类变量对于以下情况会很有用：

- 要维护所创建的所有对象的数目，或者正在使用的对象数。
- 要维护所有对象的一个列表，以便对这个列表迭代处理。
- 要存储类范围调试方法使用的一个日志文件的名称或文件描述符。
- 要维护汇总数据，如某一天一个网段所有ATM机支取的现金总额。
- 要跟踪记录类创建的最后一个对象，或者最常访问的对象。
- 维护内存中对象（由持久内存重构）的一个缓存。
- 要提供一个逆查找表，从而可以根据某个属性的值查找一个对象。

归根结底，接下来的问题是要确定在哪里存储这些共享属性的状态。相对于实例属性而言，Perl并没有特别的语法机制来声明类属性。Perl为开发人员提供了大量强大而灵活的特性，可以做一些特定的改造，以适应具体情况的特定要求。然后可以选择对给定情况最合适的机制，而不是依赖于其他的设计决定。或者，你也可以依赖于其他人打包并放在CPAN上的设计决定。重申一次，TMTOWTDI。

与类中的其他成员类似，不能直接访问类数据，特别是从类实现之外访问。如果只是为实例变量建立了严格受控的存取方法，而允许大家随意摆弄你的类变量，如设置

`$SomeClass::Debug = 1`，这可不是封装的本来目的。要在接口和实现之间建立一个清晰的防火墙，与管理实例数据的方法类似，可以创建存取方法来管理类数据。

假设我们希望跟踪记录Criticter对象的总数量。我们把这个数存储在一个包变量中，不过提供了一个名为population的方法，这样类的用户就不必知道具体实现。

```
Criticter->population()      # 通过类名访问
$gollum->population()        # 通过实例访问
```

由于Perl中的类只是一个包，要存储类数据，最自然的就是把它存储在一个包变量中。下面给出这样一个类的简单实现。population方法忽略其调用者，直接返回包变量\$Population的当前值（有些程序员喜欢把全局变量首字母大写）。

```
package Critter;
our $Population = 0;
sub population { return $Population }
sub DESTROY { $Population-- }
sub spawn {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    $Population++;
    return bless { name => shift || "anon" }, $class;
}
sub name {
    my $self = shift;
    $self->{name} = shift if @_;
    return $self->{name};
}
```

如果想让类数据方法与实例数据的存取方法类似，可以这样做：

```
our $Debugging = 0;      # 类数据
sub debug {
    shift;                # 有意地忽略调用者
    $Debugging = shift if @_;
    return $Debugging;
}
```

现在可以通过类设置全局调试级别，也可以通过它的任何实例来设置。

由于\$Debugging是一个包变量，它是全局可访问的。不过如果把our变量改为my变量，那么只有同一个文件中后面的代码可以看到这个变量。还可以更进一步，甚至可以限制类的其余部分也不能不加约束地自由访问类属性。把变量声明包围在一个块作用域中：

```
{
    my $Debugging = 0;      # 词法作用域类数据
    sub debug {
        shift;              # 有意忽略调用者
        $Debugging = shift if @_;
        return $Debugging;
    }
}
```

```
}
```

现在，除非使用存取方法，任何人都不允许读写类属性，因为只有那个子例程与变量在同一个作用域中，因此它可以访问这个变量。

如果一个派生类继承了这些类存取方法，它们仍访问原来的数据，而不论变量用`our`还是`my`声明。数据并不与包关联。可以认为这些存取方法是在定义方法（而不是调用方法）的类中执行。

对于某些类数据，这种做法很好，但对于另外一些数据就不太好了。假设我们创建了`Critter`的一个子类`Warg`。如果想保持数量分开，`Warg`不能继承`Critter`的`population`方法，因为这个方法总是返回`$Critter::Population`的值。

必须根据具体情况来确定类属性是否与包关联。如果你希望有与包关联的属性，可以使用调用者的类找到包含这个类数据的包：

```
sub debug {
    my $invocant = shift;
    my $class = ref($invocant) || $invocant;
    my $varname = $class . "::Debugging";
    no strict "refs"; # 以符号方式访问包数据
    $$varname = shift if @_;
    return $$varname;
}
```

我们临时取消了严格引用，否则我们无法使用包全局变量的完全限定符号名。这是很有道理的：因为根据定义所有包变量都存储在包中，所以通过包的符号表来访问并没有什么不对。

另一种方法是通过对象（或作为参数传入）来得到它需要的所有一切（甚至包括全局类数据）。为此，通常需要为每个类建立一个专用的构造函数，或者至少有一个专用的初始化例程供构造函数调用。在这个构造函数或初始化方法中，将类数据的引用直接存储在对象中，所以无需再查找。存取方法可以使用对象来找到数据的引用。

不用把查找类数据的复杂性放到各个方法中，只需要让对象告诉方法数据放在哪里。只有当类数据存取方法作为实例方法调用时这种做法才有效，因为类数据可能在无法访问的词法作用域变量中，只使用包名可能无法获得。

不管怎样，与包关联的类数据总有点难缠。继承一个类数据存取方法时，如果还能继承它访问的状态数据，就会清晰得多。参见`perltoot`手册页，其中给出了大量更灵活的方法，可以创造性地管理类数据。不过，你可能先要找到这个手册页。



# Moose模块

我们已经介绍了Perl的内置对象系统，不过Perl程序员还可能喜欢另一个对象系统。Moose模块使用元对象编程，可以为你做很多有意思的工作。关于Moose的内容有很多，绝不是我们在这本书里能够说清的（实际上，这本身就需要整本书来介绍），不过，可以先简单地了解一下：

```
use v5.14;

package Stables 1.01 {
    use Moose;

    has "animals" => (
        traits => ["Array"],
        is      => "rw",
        isa      => "ArrayRef[Animal]",
        default => sub { [] },
        handles => {
            add_animal => "push" ,
            add_animals => "push" ,
        },
    );

    sub roll_call {
        my($self) = @_;

        for my $animal ($self->animals) {
            say "Some ", $animal->type,
                " named ", $animal->name,
                " is in the stable";
        }
    }
}

package Animal 1.01 {
    use Moose;

    has "name" => (
        is      => "rw",
        isa      => "Str",
        required => 1,
    );

    has "type" => (
        is      => "rw",
        isa      => "Str",
        default => "animal",
    );
}

my $stables = Stables->new;

$stables->add_animal(
    Animal->new(name => "Mr. Ed", type => "horse")
)
```

```
);

$stables->add_animals(
    Animal->new(name => "Donkey", type => "donkey"),
    Animal->new(name => "Lampwick", type => "donkey"),
    Animal->new(name => "Trigger", type => "horse" ),
);
$stables->roll_call;
```

Moose做了很多事情，可以让设计类的人更轻松。Moose在Stables包中提供了一些特性，如果没有这些特性，很多的工作都要你自己来实现，这会很烦人。调用has定义了特定属性的存取方法。

### 有默认参数的默认构造函数

Stables或Animal中没有显式的构造函数。这些工作都由Moose负责。如果你需要特殊的构造函数，也可以提供你自己的构造函数。在Animal中，name属性是必需的，不过type属性有一个默认值。

### 参数检查

在Stables的has animals行中，值类型声明为包含Animal对象的ArrayRef。default指定了构造函数没有参数（因为required为0）时会做什么。Moose会检查为add\_animals提供的参数是否是一个Animal对象。

### traits

traits键为存取方法提供行为。由于这个值是一个数组引用，你可能想对它完成数组操作。handles散列引用把你想使用的名字映射到trait提供的方法名。add\_animal和 add\_animals方法会分派到Array trait的push。

这只是一个简单的例子。Moose还有更多强大功能，可以做更有意义的事情。要更多地了解Moose，可以先从它的网站开始（<http://moose.perl.org>）。

还有另外一些模块提供了类似Moose的接口。Mouse框架是Moose的一个缩减版本，旨在改善性能问题，其中去除了你不想要的特性。Moo也是一个缩减版本的Moose，不再包括部署所要求的XS前置条件。Mo框架是一个更简短的版本。

## 小结

这就是对象的全部内容了，尽管还算不上面面俱到。现在你只需要去买一本关于面向对象设计方法论的书，然后在接下来6个月里好好研读那本书。

对象固然很酷，不过有时它们实在有点太酷了。有些情况下，你可能更希望它们不那么像对象，而最好更像是常规数据类型。不过这里有一个问题：对象是引用表示的指示对象，而引用除了作为引用以外并没有太大用途。你不能让引用相加，也不能打印引用，甚至试图对引用应用Perl的内置操作符也是不行的。你唯一能做的就是解引用。所以你会写很多类似下面的显式方法调用：

```
print $object->as_string;  
$new_object = $subject->add($object);
```

这种显式解引用总的来讲是好事。绝对不要混淆引用和指示对象，除非你有意混淆它们的差别。下面就是这样一种情况。如果利用重载（overloading）设计类，可以假装没有引用，简单地写为：

```
print $object;  
$new_object = $subject + $object;
```

重载Perl的某个内置操作符时，需要定义将这个操作符应用到某个类的对象时会有怎样的行为。很多标准Perl模块都使用了重载，如Math::BigInt，它允许你创建Math::BigInt对象，这些对象的表现就像是常规的整数，但是没有大小限制。可以用+将Math::BigInt对象相加，用/将它们相除，用<=>来进行比较，还可以用print打印。

需要注意，重载与自动加载（autoloading）是不一样的，自动加载是指根据需要自动地加载一个缺少的函数或方法。重载与覆盖（overriding）也不同，覆盖是一个函数或方法掩盖另一个函数或方法。重载什么都不隐藏。它只是为一个操作增加新的含义，否则对引用完成这个操作没有任何意义。



# overload Pragma

`overload pragma`实现操作符重载。需要为它提供操作符及其相关行为的一个键/值列表（操作符作为键，相关行为作为值）：

```
package MyClass;

use overload "+" => \&myadd,          # 代码引用
              "<" => "less_than",      # 命名方法
              "abs" => sub { return @_ }; # 匿名子例程
```

现在如果你想将两个MyClass对象相加，就会调用myadd子例程来生成结果。

如果试图用<操作符比较两个MyClass对象，Perl会注意到这个行为指定为一个字符串，并把这个字符串解释为一个方法名，而不只是一个子例程名。在上面的例子中，less\_than方法可能由MyClass包自己提供，也可能从MyClass的一个基类继承，不过myadd子例程必须由当前包提供。对应abs的匿名子例程甚至更直接，它会自行提供。不论这些子例程以何种方式提供，我们把它们都称为处理方法（handlers）。

对于一元操作符（只取一个操作数，如abs），只要这个操作符应用到某个类的对象，就会调用为该类指定的处理方法。

对于二元操作符，如+或<，只要第一个操作数是某个类的对象，或者当第二个操作数是该类的对象而且第一个操作数没有重载行为，就会调用为该类指定的处理方法。所以可以写为：

```
$object + 6
```

或者：

```
6 + $object
```

而不用担心操作数的顺序（在第二种情况中，操作数传递到处理方法时会交换顺序）。如果表达式写为：

```
$animal + $vegetable
```

\$animal和\$vegetable是不同类的对象，它们都使用了重载，这样就会触发\$animal的重载行为（我们希望动物喜欢蔬菜）。

Perl中只有一个三元操作符?:，很幸运，这个操作符不能重载。

## 重载处理方法

一个重载的操作符执行操作时，会调用相应的处理方法并提供3个参数。前面两个参数是两个操作数。如果这个操作符只使用一个操作数，第二个参数则是undef。

第三个参数指示前两个参数是否交换顺序。即使按照常规的算术运算规则，有些操作（如加法或乘法）并不关心参数的顺序，不过另外一些操作则不然，如减法和除法，它们确实很在乎参数的顺序<sup>注1</sup>。考虑下面两种写法的区别：

```
$object - 6
```

和

```
6 - $object
```

如果处理方法的前两个参数交换了顺序，第三个参数则为true。否则，第三个参数为false，在这种情况下，还可以更细地区分：如果处理方法由另一个涉及赋值的处理方法触发（如+=中使用+来确定如何相加），那么第三个参数不仅仅是false，而是undef。利用这个区别可以做一些优化。

举例来说，以下给出一个类，允许你管理一个固定范围内的数字。这个类重载了+和-，对象相加或相减的结果值会限制在0到255的范围内：

```
package ClipByte;

use overload "+" => \&clip_add,
              "-" => \&clip_sub;

sub new {
    my $class = shift;
    my $value = shift;
    return bless \$value => $class;
}

sub clip_add {
    my ($x, $y) = @_;
    my ($value) = ref($x) ? $$x : $x;
    $value += ref($y) ? $$y : $y;
    $value = 255 if $value > 255;
    $value = 0 if $value < 0;
    return bless \$value => ref($x);
}

sub clip_sub {
    my ($x, $y, $swap) = @_;
    my ($value) = (ref $x) ? $$x : $x;
    $value -= (ref $y) ? $$y : $y;
    if ($swap) { $value = -$value }
    $value = 255 if $value > 255;
    $value = 0 if $value < 0;
    return bless \$value => ref($x);
}
```

---

注1：当然，并不要求重载的对象遵循常规的算术运算规则，不过最好不要让人感觉太怪异。奇怪的是，很多语言都犯了一个错误，总是用字符串连接操作重载+，要知道字符串连接没有交换性，只能含糊地相加。可以参考Perl来了解一种不同的做法。

```

package main;

$byte1 = ClipByte->new(200);
$byte2 = ClipByte->new(100);

$byte3 = $byte1 + $byte2;      # 255
$byte4 = $byte1 - $byte2;      # 100
$byte5 = 150 - $byte2;         # 50

```

可以注意到，这里的每一个函数都必然是构造函数，所以每个函数要负责祝福（ `bless` ）其新对象，将它重新转换为当前类（不论当前类是什么），这里假设我们的类可以继承。另外还假设如果 `$y` 是一个引用，那么它指向一个对象，而且对象类型是我们自己的类型。可以不用测试 `ref($y)`，如果想更彻底地测试，可以调用 `$y->isa("ClipByte")`（当然运行得更慢）。

## 可重载操作符

只能重载某些操作符，如表13-1所示。如果有 `use overload` 声明，这些操作符也会列在 `%overload::ops` 散列中，不过类别划分与这里稍有不同。

表13-1：可重载操作符

类别	操作符
转换操作符	<code>"0+ bool qr</code>
算术操作符	<code>+ - * / % ** x . neg</code>
逻辑操作符	<code>!</code>
位操作符	<code>&amp;   ~ ^ ! &lt;&lt; &gt;&gt;</code>
赋值操作符	<code>+= -= *= /= %= **= x= .= &lt;&lt;= &gt;&gt;= ++ --</code>
比较操作符	<code>== &lt; &lt;= &gt; &gt;= != &lt;=&gt; lt le gt ge eq ne cmp</code>
数学函数	<code>atan2 cos sin exp abs log sqrt int</code>
迭代操作符	<code>&lt;&gt;</code>
文件测试操作符	<code>-X</code>
解引用操作符	<code>\${} @{} %{} &amp;{} *{}</code>
匹配操作符	<code>~~</code>
伪操作符	<code>nomethod fallback =</code>

需要说明，`neg`、`bool`、`nomethod`和`fallback`都不是真正的Perl操作符。5个解引用操作符、`qr`、`"`和`0+`看起来可能也不像操作符。不过，它们都是合法的键，可以放在提供给 `use overload` 的参数表中。这没有问题。告诉你一个小秘密：说 `overload pragma` 重载操作符实际上有点骗人。它重载的是底层操作，而不论是通过其“官方”操作符显式调用，



还是通过某个相关的操作符隐式调用（我们提到过的伪操作符只能隐式调用）。换句话说，重载并不是发生在语法层，而是发生在语义层。关键不是要好看，而是要正确。这个道理还可以推广。

还要注意，`=`不会像你预想的那样重载Perl的赋值操作符。这样是不对的。稍后还会更详细地说明。

我们首先讨论转换操作符，并不是因为它们最明显（实际上它们并不明显），而是因为它们最有用。很多类只重载了字符串化（由`""`键指定）。（没错，这确实是两个连续的双引号）。

转换操作符：`""`，`0+`，`bool`，`qr`

前面3个键分别允许Perl自动转换为字符串、数字和布尔值。

第4个键用于对象内插到一个正则表达式或者用作为正则表达式时（包括作为`=~`或`!~`操作符的右操作数出现）。`qr`子例程必须返回一个已编译的正则表达式，或者返回一个已编译正则表达式（如由真正的`qr`返回）的引用，对返回值的进一步重载都将被忽略。

我们说任何非字符串变量用作为字符串时会出现“字符串化”（stringification）。如果通过打印、内插、连接或者甚至用作为一个散列键将一个变量转换为字符串，就会发生这种字符串化。正是因为这种字符串化，试图打印一个对象时你可能会看到类似`SCALAR(0xba5fe0)`的结果。

在数值上下文中，如任意数学表达式、数组索引或者甚至作为`..`（范围操作符）的操作数，非数值变量转换为一个数字时会出现“数字化”（numification）。

最后一点，尽管没有人把它叫做“布尔化”（boolification），不过通过创建一个`bool`处理方法，可以定义对象在一个布尔上下文中（如`if`、`unless`、`while`、`for`、`and`、`or`、`&&`、`||`、`?:`或`grep`表达式中的代码块）应当如何解释。

如果有任意一个转换操作符，这几个转换操作符都能自动生成，稍后我们还会解释自动生成（autogeneration）。你的处理方法可以返回你喜欢的任何值。需要说明，如果触发转换的那个操作也被重载，该重载就会随即发生。

下面的例子展示了`""`在字符串化时调用了对象的`as_string`处理方法。不要忘记对引号加引号：

```
package Person;

use overload q("") => \&as_string;

sub new {
    my $class = shift;
    return bless { @_ } => $class;
}
```

```

sub as_string {
    my $self = shift;
    my ($key, $value, $result);
    while (($key, $value) = each %$self) {
        $result .= "$key => $value\n";
    }
    return $result;
}

$obj = Person->new(height => 72, weight => 165, eyes => "brown");

print $obj;

```

并不是输出类似`Person=HASH(0xba1350)`的结果，这会（按散列顺序）打印以下结果：

```

weight => 165
height => 72
eyes => brown

```

（我们真心希望这个人不是体重165kg而身高72cm）。

**算术操作符：**`+`，`-`，`*`，`/`，`%`，`**`，`x`，`.`，`neg`

除了`neg`外，这些算术操作符对你来说应该都很熟悉，`neg`是用于一元负号（如-123中的`-`）的一个特殊重载键。利用`neg`和`-`的区别，你可以为一元负号和二元减号指定不同的行为，二元减号通常表示减法。

如果重载`-`而没有重载`neg`，然后试图使用一元负号，Perl会为你模拟一个`neg`处理方法。这称为自动生成（autogeneration），发生自动生成的情况下，某些操作符可以从其他操作符合理地推导得出（前提是重载操作符之间与常规操作符之间有相同的关系）。由于一元负号可以表示为二元减号的一个函数（也就是说，-123等价于`0 - 123`），所以`-`重载时，Perl并不强制重载`neg`（当然，如果你故意把二元减号定义为用第一个参数除第二个参数，一元负号就很可能抛出一个“除0”异常）。

利用`.`操作符实现的连接可以通过字符串化处理方法自动生成（参见上面“转换操作符”小节中的`"`）。

**逻辑操作符：**`!`

如果没有为`!`指定处理方法，可以使用`bool`、`"`或`0+`处理方法自动生成`!`的处理方法。如果重载了`!`操作符，`not`操作符也会触发你请求的任何行为（还记得我们的小秘密吗）。

你可能会很奇怪：怎么没有其他逻辑操作符？实际上，大多数逻辑操作符都不能重载，这是因为它们是短路的。它们是真正的控制流操作符，需要能够延迟某些参数的计算。正是因为这个原因，`?:`操作符也不能重载。

位操作符: &, |, ~, ^, <<, >>

~操作符是一个一元操作符, 所有其他位操作符都是二元操作符。下面的例子展示了可以重载>>完成类似chop的工作:

```
package ShiftString;

use overload
    ">>" => \&right_shift,
    q("") => sub { ${ $_[0] } };

sub new {
    my $class = shift;
    my $value = shift;
    return bless \$value => $class;
}

sub right_shift {
    my ($x, $y) = @_;
    my $value = $$x;
    substr($value, -$y) = "";
    return bless \$value => ref($x);
}

$camel = ShiftString->new("Camel");
$ram = $camel >> 2;
print $ram;          # Cam
```

赋值操作符: +=, -=, \*=, /=, %=, \*\*=, x=, .=, <<=, >>=, ++, --

这些赋值操作符可以改变其参数的值, 也可以保留原值。只有当新值与原值不同时才会把结果赋给左操作数。这样就可以使用同一个处理方法来重载+=和+。尽管这是允许的, 不过通常并不建议这么做, 因为根据后面“缺少重载处理方法时 (no-method和fallback)”一节介绍的语义, Perl总会调用+的处理方法, 而假设+=没有直接重载。

连接赋值 (.=) 可以使用字符串化再加一个正常的字符串连接自动生成。++和--操作符可以由+和- (或+=和-=) 自动生成。

实现++和--的处理方法往往会改变 (修改) 其参数。如果希望自增操作不仅用于数字, 还可以作用于字母, 可以用以下处理方法实现:

```
package MagicDec;

use overload
    q(-- ) => \&decrement,
    q("") => sub { ${ $_[0] } };

sub new {
    my $class = shift;
    my $value = shift;
    bless \$value => $class;
}
```



```

sub decrement {
    my @string = reverse split(//, ${ $_[0] } );
    my $i;
    for ($i = 0; $i < @string; $i++) {
        last unless $string[$i] =~ /a/i;
        $string[$i] = chr( ord($string[$i]) + 25 );
    }
    $string[$i] = chr( ord($string[$i]) - 1 );
    my $result = join(" " => reverse @string);
    $_[0] = bless \$result => ref($_[0]);
}

package main;

for $normal (qw/perl NZ Pa/) {
    $magic = MagicDec->new($normal);
    $magic--;
    print "$normal goes to $magic\n";
}

```

这会打印以下结果：

```

perl goes to perk
NZ goes to NY
Pa goes to Oz

```

它的作用与Perl的魔法字符串自增操作符完全相反。

`++$a`操作可以使用`$a += 1`或`$a = $a + 1`自动生成，`$a--`可以使用`$a -= 1`或`$a = $a - 1`自动生成。不过，这不会像真正的`++`操作符那样触发复制行为。参见这一章后面“复制构造函数(=)”一节的介绍。

**比较操作符：**`==`, `<`, `<=`, `>`, `>=`, `!=`, `<=>`, `lt`, `le`, `gt`, `ge`, `eq`, `ne`, `cmp`

如果重载了`<=>`，可以用它来自动生成`<`, `<=`, `>`, `>=`, `==`和`!=`的行为。类似的，如果重载了`cmp`，则可以用它自动生成`lt`, `le`, `gt`, `ge`, `eq`和`ne`的行为。

需要说明，即使重载了`cmp`，也不能像你想象中那么容易地对对象排序，因为所比较的是字符串化的对象，而不是对象本身。如果你要比较实际的对象，那么还需要重载`""`。

**数学函数：**`atan2`, `cos`, `sin`, `exp`, `abs`, `log`, `sqrt`, `int`

如果`abs`不可用，可以由`<`或`<=>`并结合一元负号或二元减号来自动生成。

可以用重载的`-`为一元负号或`abs`函数自动生成缺少的处理方法，它们也可以重载（没错，我们知道`abs`看起来像一个函数，而一元负号看起来像一个操作符，不过对Perl来说，它们的差别并没有看上去那么大）。

按照传统，Perl函数`int`会向0取整（参见第27章的`int`条目），所以对于表现得像浮点类型的对象，可能要做同样的处理，以免让人困惑。

迭代操作符: <>

通过使用`readline` (从一个文件句柄读取, 如`while (<FH>)`) 或者`glob` (用于文件  
团匹配, 如`@files = <*. *>`), 可以触发<>处理方法:

```
package LuckyDraw;

use overload
    "<>" => sub {
        my $self = shift;
        return splice @$self, rand @$self, 1;
    };

sub new {
    my $class = shift;
    return bless [ @_ ] => $class;
}

package main;

$lotto = new LuckyDraw 1 .. 51;

for (qw(1st 2nd 3rd 4th 5th 6th)) {
    $lucky_number = <$lotto>;
    print "The $_ lucky number is: $lucky_number.\n";
}

$lucky_number = <$lotto>;
print "\nAnd the bonus number is: $lucky_number.\n";
```

如果在加利福尼亚, 这会打印以下结果:

```
The 1st lucky number is: 18
The 2nd lucky number is: 11
The 3rd lucky number is: 40
The 4th lucky number is: 7
The 5th lucky number is: 51
The 6th lucky number is: 33
And the bonus number is: 5
```

### 文件测试操作符

键-X用于指定一个子例程来处理所有文件测试操作符, 如-f、-x等。参见第3章“命名一元操作符和文件测试操作符”一节中的表3-4。

不能单独地重载某一个文件测试操作符。为了区别不同的文件测试操作符, -后面的字母要作为第二个参数传入 (也就是说, 传入二元操作符用来传递第二个操作数的槽中)。

调用一个重载的文件测试操作符不会影响与特殊文件句柄\_关联的`stat`值。它仍指向上一个`stat`、`lstat`或未重载文件测试的结果。

文件测试操作符的重载在v5.12中引入。

解引用操作符：\${}，@{}，%{}，&{}，\*{}

如果试图解引用标量、数组、散列、子例程和类型团引用，可以通过重载这5个符号来截取。

overload的Perl在线文档展示了如何使用这个操作符来模拟你自己的伪散列。这里给出一个更简单的例子，我们将对象实现为一个匿名数组，不过允许散列引用。不要把它当作一个真正的散列，你不能从这个对象删除键/值对。如果你想结合数组和散列记法，就要使用真正的伪散列（可以这么说）。

```
package PsychoHash;

use overload "%{}" => \&as_hash;

sub as_hash {
    my ($x) = shift;
    return { @$x };
}

sub new {
    my $class = shift;
    return bless [ @_ ] => $class;
}

$critter = new PsychoHash( height => 72, weight => 365, type => "camel" );
print $critter->{weight}; # 打印365
```

参见第14章，其中介绍的一种机制允许你重新定义散列、数组和标量的基本操作。

重载一个操作符时，不要创建包含自身引用的对象。例如：

```
use overload "+" => sub { bless [ $_[0], $_[1] ] };
```

这是在自找麻烦，因为如果有`$animal += $vegetable`，其结果将是`$animal`作为一个引用，它指向一个被祝福的数组引用，而该数组中的第一个元素是`$animal`。这是循环引用（circular reference），这意味着即使你撤销了`$animal`，在进程（或解释器）终止之前`$animal`的内存都不会释放。

参见第8章“垃圾回收、循环引用和弱引用”一节。

### 智能匹配

键`~~`允许你覆盖`~~`操作符和`given`构造使用的智能匹配逻辑。参见第3章的“智能匹配操作符”和第4章的“`given`语句”小节。

在一些不常见的情况下，智能匹配操作符的重载实现并没有全面控制智能匹配行为。具体地，在以下代码中：

```
package Foo;
use overload "~~" => "match";

my $obj = Foo->new();
$obj ~~ [ 1,2,3 ];
```



这个智能匹配并没有做出这样的方法调用：

```
$obj->match([1,2,3],0); # 不正确的调用
```

实际上，这里会优先考虑智能匹配分配律，所以会把\$obj依次与各个数组元素智能匹配，直到找到一个匹配为止，所以最后实际上你可能会看到1到3个以下的调用：

```
$obj->match(1,0);  
$obj->match(2,0);  
$obj->match(3,0);
```

关于何时调用智能匹配操作符的重载行为，详细内容参见第3章的表3-7。

## 复制构造函数(=)

尽管=看起来很像一个常规的操作符，不过作为一个重载键，它的含义有些特殊而且不太直观。它并不重载Perl赋值操作符，也不能重载那个操作符，因为这个操作符要保留用来完成赋值引用，否则一切都会出问题。

如果对一个引用应用了一个修改方法（如++、--或任何赋值操作符），而且该引用与另一个引用指示同一个对象，这种情况下就会使用=的处理方法。= 处理方法允许你拦截这个修改方法，由你自己复制对象，这样就能独立地修改这个副本。否则将会破坏原对象。

```
$copy = $original;    # 只复制引用  
++$copy;              # 修改底层共享对象
```

下面我们继续。假设\$original是指向一个对象的引用。要让++\$copy只修改\$copy而不是修改\$original，首先建立\$copy的一个副本，然后将指向这个新对象的引用赋给\$copy。直到++\$copy执行时才会完成这个操作，所以在自增之前\$copy碰巧与\$original是一样的。不过，自增之后它们就不同了。换句话说，++会发现需要复制并调用你的复制构造函数。

只有++或+=等修改方法或者nomethod（稍后介绍）才能识别是否需要复制。如果操作通过+自动生成，如：

```
$copy = $original;  
$copy = $copy + 1;
```

就不会出现复制，因为+不知道它要用作为一个修改方法。

如果某个修改方法的执行期间需要调用复制构造函数，但是并没有为=指定处理方法，倘若对象只是一个平常的标量而不是更复杂的类型，=可以自动生成为一个字符串复制操作。

例如，对于以下代码序列：

```
$copy = $original;
...
++$copy;
```

实际执行的代码可能如下：

```
$copy = $original;
...
$copy = $copy->clone(undef, "");
$copy->incr(undef, "");
```

这里假设\$original指向一个重载的对象，++重载为\&incr，=重载为\&clone。

\$copy = \$original++也会触发类似的行为，这会解释为\$copy = \$original; ++\$original。

## 缺少重载处理方法时（nomethod和fallback）

如果对一个对象应用一个未重载的操作符，Perl首先会用之前介绍的规则尝试从其他重载操作符自动生成一个行为。如果失败，Perl会寻找nomethod的重载行为，如果找到则使用这个重载行为。这个处理方法对于操作符来说就像是AUTOLOAD子例程对于其他子例程：如果没有其他选择，就可以用它。

如果使用nomethod，nomethod键后面必须跟一个处理方法的引用，这个处理方法接受4个参数（而所有其他处理方法都接受3个参数）。前3个参数与其他处理方法的参数并没有不同；第4个参数是一个字符串，对应于这个缺少处理方法的操作符。它与\$AUTOLOAD变量在AUTOLOAD子例程中的作用是一样的。

如果Perl必须查找一个nomethod处理方法，但是找不到，就会产生一个异常。

如果你想避免发生自动生成，或者你希望自动生成失败而导致根本不重载，可以定义另外一个特殊的fallback重载键。它有3个很有用的状态：

### undef

如果fallback未设置，或者显式地设置为undef，重载事件序列则不受影响：总是先查找处理方法，尝试自动生成，最后调用nomethod处理方法。如果失败，则产生一个异常。

### false

如果fallback设置为一个确定值，但是为false（如0），则不会尝试自动生成。如果存在nomethod处理方法，Perl就会调用这个方法，不过如果不存在这个处理方法，则产生一个异常。

true

这与undef的行为几乎是一样的，不过如果未能通过自动生成合成一个合适的处理方法，并不会产生异常。相反，Perl仍会采用该操作符的未重载的行为，就好像类中根本没有use overload pragma一样。

## 重载常量

可以用overload::constant改变Perl以何种方式解释常量，最有用的做法是把overload::constant放在包的import方法中（如果这样做，一定要在这个包的unimport方法中适当地调用overload::remove\_constant，这样一来，如果你有要求，包可以自行完成清理工作）。

overload::constant和overload::remove\_constant都需要一个键/值对列表。键应当是integer、float、binary、q和qr，而且各个值应当是子例程名、匿名子例程或者是将处理这些常量的一个代码引用。

```
sub import { overload::constant ( integer => \&integer_handler,  
                                  float   => \&float_handler,  
                                  binary  => \&base_handler,  
                                  q       => \&string_handler,  
                                  qr      => \&regex_handler ) }
```

只要Perl词法分析器遇到常量数字，就会调用你为integer和float提供的处理方法。这一点独立于constant pragma；类似下面的简单语句：

```
$year = cube(12) + 1;      # 整数  
$pi = 3.14159265358979;   # 浮点数
```

会触发你请求的任何处理方法。

binary键允许截取二进制、八进制和十六进制常量。q处理单引号字符串（包括用q引入的字符串）以及qq-和qx-引用字符串和here文档中的常量子串。最后，qr处理正则表达式中的常量部分，参见第5章最后部分的介绍。

要为处理方法传入3个参数。第一个参数是原常量，仍采用提供给Perl的原有形式。第二个参数是Perl实际上如何解释这个常量，例如，123\_456将显示为123456。

只会为由q和qr处理方法处理的字符串定义第3个参数，可能是qq、q、s或tr，取决于字符串将如何使用。qq表示这个字符串来自一个内插上下文，如双引号、重音符、m//匹配或s///替换的模式。q表示这个字符串来自一个非内插上下文，s表示常量是s///替换中的一个替换字符串，tr表示它是tr///或y///表达式的一部分。



处理方法应当返回一个标量，它将取代常量。通常情况下，这个标量是一个重载对象的引用，不过你完全可以更谨慎一些：

```
package DigitDoubler;          # 将放在DigitDoubler.pm中的一个模块
use overload;

sub import { overload::constant ( integer => \&handler,
                                   float   => \&handler ) }

sub handler {
    my ($orig, $interp, $context) = @_ ;
    return $interp * 2;          # 将所有常量加倍
}

1;
```

需要说明，这个处理方法由两个键共享，在这种情况下这是可以的。现在如果有以下代码：

```
use DigitDoubler;

$trouble = 123;      # trouble现在是246
$jeopardy = 3.21;    # jeopardy现在是6.42
```

你已经重新定义了整个世界。

如果截取字符串常量，建议你还要提供一个连接操作符（“.”），因为类似“ab\$cd!!”的内插表达式只是更长的“ab”. \$cd . “!!”表达式的一个简写形式。类似的，负数被认为是正常量的负值，所以如果截取整数或浮点数，就应该为neg提供一个处理方法（之前我们不需要这么做，因为我们只是返回实际的数字，而不是重载的对象引用）。

要注意，overload::constant不会传播到eval中的运行时编译，这可以说是一个bug，也可以说是一个特性，这要看你怎么理解。

## 公共重载函数

在Perl v5.6中，overload pragma提供了以下可以公共使用的函数。

**overload::StrVal(OBJ)**

这个函数返回没有字符串化重载（“”）时OBJ会有的字符串值。

**overload::Overloaded(OBJ)**

如果OBJ应用了操作符重载，这个函数会返回一个true值，否则返回false。

**overload::Method(OBJ, OPERATOR)**

这个函数为处理OBJ的操作符返回其重载实现代码的一个引用，如果不存在这种重载，则返回undef。

## 继承和重载

继承与重载有两种交互方式。如果处理方法命名为一个字符串，而不是提供为一个代码引用或匿名子例程，就会采用第一种交互方式。命名为一个字符串时，处理方法会解释为一个方法，因此可以从超类继承。

继承和重载之间还有第二种交互：从一个重载类派生的任何类本身也有这种重载行为。换句话说，重载本身是继承的。一个类中的处理方法集合是该类所有祖先的处理方法的并集。如果一个处理方法可能在多个不同的祖先类中找到，实际使用的处理方法由方法继承的通用原则来确定。例如，如果类Alpha继承自类Beta和Gamma（顺序是先Beta，后Gamma），而且类Beta将+重载为\&Beta::plus\_sub，不过类Gamma将+重载为字符串"plus\_meth"，那么试图对一个Alpha对象应用+时会调用Beta::plus\_sub。

由于fallback键的值不是一个处理方法，其继承不受上述规则的控制。在当前实现中，会使用第一个重载祖先的fallback值，不过这有偶发性，可能会在不加说明的情况下改变（嗯，更确切地讲是可能没有太多说明）。

## 运行时重载

由于use语句在编译时执行，所以要想在运行时改变重载，唯一的办法是：

```
eval " use overload '+' => \&my_add ";
```

还可以写为：

```
eval " no overload '+', '--', '<=' ";
```

不过在运行时使用这种构造是有问题的。

## 重载诊断

编译Perl时如果有-DDEBUGGING选项，那么运行程序时倘若加上-Do开关或类似选项，你就能看到重载的诊断消息。还可以使用Perl内置调试器的m命令推断哪些操作得到了重载。

如果你感觉内容太多，负荷太重，可能下一章可以给你提供一些约束。

# 绑定变量

有些人类活动需要伪装。有时是为了欺骗，不过更多的时候是为了更深层次更真实地交流。例如，很多面试官希望你穿西装打领带来表示你确实适合这个工作，尽管你们都知道在实际工作中你是不会打领带的。想想看也真是很奇怪：只是在脖子上围一块布就能神奇地帮你找到工作。在Perl文化中，`tie`操作符也有类似的作用：它允许你创建一个看起来很平常的变量，不过在这个伪装之下，这实际上是一个完整的Perl对象，有它自己的有趣个性。这是一个有点奇怪的小魔术，就像从帽子里弹出一个邦尼兔。

换种方式来看，变量名前面的趣味字符`$`、`@`、`%`或`*`能告诉Perl和Perl程序员很多信息，它们分别暗示一组特定的原型行为。通过将变量与实现一组新行为的类关联，可以采用各种有用的方式用`tie`包装这些行为。例如，可以创建一个常规的Perl散列，然后将它绑定（`tie`）到一个类，这个类将散列放在一个数据库中，这样一来，以后从散列读取值时，Perl就会魔法般地从外部数据库文件获取数据，而当你设置散列中的值时，Perl又会魔法般地将数据存储到这个外部数据库文件中。在这里，“魔法般地”表示“透明地完成一些非常复杂的工作”。你肯定听过一句老话：再先进的技术与Perl脚本相比也没有什么区别（说实在的，研究Perl核心的人总把“魔法”（`magic`）一词当作一个技术术语，表示为变量附加的一些额外的语义，如`%ENV`或`%SIG`。绑定变量就是这种魔法的一个延伸）。

Perl已经提供了内置`dbmopen`和`dbmclose`函数，可以将散列变量魔法般地绑定到数据库，不过这些函数是在Perl还没有`tie`时实现的。而现在，`tie`提供了一种更通用的机制。实际上，`dbmopen`和`dbmclose`就是Perl利用`tie`的机制实现的。

可以把一个标量、数组、散列或文件句柄（通过其类型团）绑定到某个类，只要它提供了有适当名字的方法，可以拦截和模拟对这些变量的正常访问。在这些方法中，第一个方法是绑定（`tie`）发生时调用的：绑定一个变量总会调用一个构造函数，如果成功，这会返回



一个对象，Perl会把这个对象藏在你看不到的地方，它会深深地藏在“普通”变量里面。不过以后完全可以对普通变量使用tied函数来获取这个对象：

```
tie VARIABLE, CLASSNAME, LIST; # 将VARIABLE绑定到CLASSNAME
$object = tied VARIABLE;
```

这两行代码等价于：

```
$object = tie VARIABLE, CLASSNAME, LIST;
```

一旦绑定，就可以正常地处理普通变量，不过每次访问都会自动地调用底层对象的方法；类的所有复杂性都隐藏在这些方法调用中。如果以后你想断开变量和类之间的这种关联，可以解除变量的绑定（untie）：

```
untie VARIABLE;
```

几乎可以把tie看作是一种有趣的祝福（bless），只不过它祝福的是一个裸变量而不是一个对象引用。另外还需要额外的参数，就像构造函数一样。这并不奇怪，因为它实际上就是在内部调用了一个构造函数，具体的构造函数名取决于绑定变量的类型：可能是TIESCALAR、TIEARRAY、TIEHASH或者TIEHANDLE<sup>注1</sup>。这些构造函数作为类方法调用，指定的CLASSNAME作为调用者，另外还会提供LIST中的值作为附加参数（VARIABLE不会传递给构造函数）。

这4个构造函数都以平常的方式返回一个对象。它们并不关心是不是从tie调用的，类中的其他方法也不在意这一点，因为如果你愿意，完全可以直接调用。从某种意义上讲，所有魔法都发生在tie中，而不是发生在实现tie的类中。就类而言，这只是一个普通的类，只不过有一些有趣的方法名（实际上，有些绑定模块还提供了额外的方法，通过绑定变量无法看到这些方法，它们必须像其他对象方法一样显式调用。这些额外的方法可以提供类似文件锁定、事务保护等服务，或者能够完成实例方法可以做的任何其他事情）。

所有这些构造函数会祝福（bless）并返回一个对象引用，就像其他构造函数一样。这个引用指向的变量并不需要与所绑定的变量类型相同，但是它必须被祝福到某个类，以使绑定变量可以反过来找到这个类。例如，后面会给出一个很长的TIEARRAY例子，其中使用了一个基于散列的对象，这样它就可以很方便地保存所模拟数组的额外信息。

tie函数不会为你use或require一个模块。如果必要，你必须在调用tie之前显式地通过use或require自行加载模块（另一方面，为了保持向后兼容，dbmopen函数会尝试用use加载某个DBM实现。不过你可以用一个显式的use覆盖它的选择，只要你用use加载的模块也在dbmopen的模块列表中。更全面的解释参见AnyDBM\_File模块的在线文档）。

---

注1： 因为构造函数有不同的名字，所以甚至可以提供一个类实现所有这些构造函数。这样一来，就可以把标量、数组、散列和文件句柄都绑定到同一个类，不过通常并不这么做，因为这样会让其他魔法方法很难编写。

绑定变量调用的方法都有预定的名字（如FETCH和STORE），因为它们要从Perl内部隐式调用（也就是说，由特定的事件触发）。这些名字采用全大写（ALLCAPS），对于这种隐式调用的例程我们通常都遵循这种约定。（遵循这个约定的其他特殊名字还包括BEGIN、CHECK、UNITCHeck、INIT、END、DESTROY和AUTOLOAD，更不用说还有UNIVERSAL->VERSION。实际上，几乎Perl的所有预定义包、变量和文件句柄都采用大写：STDIN、SUPER、CORE、CORE::GLOBAL、DATA、@EXPORT、@INC、@ISA、@ARGV和%ENV。当然，内置函数和pragma要算另一个极端，它们完全没有大写字母）。

我们要介绍的第一个内容很简单：就是如何绑定一个标量变量。

## 绑定标量

要实现一个绑定标量，类必须定义以下方法：TIESCALAR、FETCH和STORE（可能还要定义UNTIE和DESTROY）。绑定一个标量变量时，Perl会调用TIESCALAR。读取一个绑定变量时，会调用FETCH。向一个变量赋值时，则会调用STORE。如果保存了初始tie返回的对象（或者如果以后使用tied获取到了这个对象），那么可以自行访问底层对象，这不会触发其FETCH或STORE方法。作为一个对象，这并不神奇，但很客观。

如果你定义了UNTIE，Perl会在解除变量绑定时调用这个方法。这就使你在这个关联消失（变量不再特殊）之前有机会做一些必要的记录或清理工作。

如果存在一个DESTROY方法，Perl会在对象的最后一个引用消失时调用这个方法，就像所有其他对象一样。这会在程序终止或调用untie时发生。untie会删除tie使用的引用，不过，untie并不会删除可能存储在其他地方的引用；直到那些引用都消失后才会调用DESTROY。

Tie::Scalar和Tie::StdScalar包都在标准Tie::Scalar模块中，如果你不想自己定义所有那些方法，可以利用这两个包，它们提供了一些简单的基类定义。Tie::Scalar提供了一些基础性方法，只能完成很少的工作，而Tie::StdScalar提供的方法可以让一个标量表现得就像一个普通的Perl标量（看起来好像很没用，不过有时你可能只是想为普通标量语义加一个包装，例如统计某个特定变量设置了多少次）。

在给出详细例子并全面介绍所有这些原理之前，下面先给你上一道开胃小菜，从中可以看到实际上这是多么容易。下面是一个完整的程序：

```
#!/usr/bin/perl
package Centsible;
sub TIESCALAR { bless \my $self, shift }
sub STORE { ${ $_[0] } = $_[1] } # 完成缺省工作
sub FETCH { sprintf "%.02f", ${ my $self = shift } } # 舍入值

package main;
```



```
tie $bucks, "Centsible";
$bucks = 45.00;
$bucks *= 1.0715; # 计税
$bucks *= 1.0715; # 双倍计税!
print "That will be $bucks, please.\n";
```

运行时，这个程序会生成以下结果：

```
That will be 51.67, please.
```

注释掉tie调用，可以看到与之前的差别。现在会得到：

```
That will be 51.66505125, please.
```

必须承认，这比通常直接舍入数字要多做一些工作。

## 标量绑定方法

从前面的例子已经看到了绑定标量有什么作用，下面来开发一个更复杂的标量绑定类。这里不使用预先完成的包作为基类（特别是因为标量实在太简单了），我们会依次介绍这4个方法，建立一个名为ScalarFile的类作为例子。绑定到这个类的标量包含普通的字符串，每一个这样的绑定变量都隐含地与一个文件关联，字符串就存储在这个文件中（可以适当地命名变量，以便你记住它指示哪一个文件）。可以采用以下方式将变量绑定到类：

```
use ScalarFile; # 加载ScalarFile.pm
tie $camel, "ScalarFile", "/tmp/camel.lot";
```

变量一旦绑定，它原来的内容就会彻底破坏，变量与其对象之间的内部关联会覆盖变量正常的语义。请求\$camel的值时，现在会读取/tmp/camel.lot的内容，为\$camel赋值时，会把新内容写到/tmp/camel.lot，而清除文件中原有的所有内容。

绑定是针对变量的，而不是针对值，所以变量的绑定性质并不会随赋值传播。例如，假设复制一个绑定的变量：

```
$dromedary = $camel;
```

并不是以平常的方式从\$camel标量变量读取值，Perl会在相关的底层对象上调用FETCH方法。就好像你写了以下代码：

```
$dromedary = (tied $camel)->FETCH();
```

或者如果你记得tie返回的对象，可以直接使用那个引用，如以下示例代码所示：

```
$clot = tie $camel, "ScalarFile", "/tmp/camel.lot";
$dromedary = $camel;           # 通过隐式接口
$dromedary = $clot->FETCH();    # 作用相同，但显式调用
```

如果除了TIESCALAR、FETCH、STORE和DESTROY之外，类还提供了其他方法，可以手动地



使用`$clot`来调用这些方法。不过，正常情况下应该自己管好自己的事情，不要去干涉底层对象，正是因为这个原因，你会经常看到从`tie`返回的值常被忽略。如果以后还需要（例如，如果类恰好记录了你需要的额外方法），仍然可以通过`tied`得到这个对象。忽略`tie`返回的对象还可以消除某些错误，这个内容我们稍后介绍。

下面是这个类的声明代码，这些代码将放在`ScalarFile.pm`中：

```
package ScalarFile;
use Carp;                # 妥善地传播错误消息
use strict;              # 自己施加一些约束
use warnings;            # 打开词法作用域警告
use warnings::register;   # 允许用户声明"use warnings 'ScalarFile'"
my $count = 0;           # 绑定ScalarFiles的内部计数
```

标准Carp模块导出`carp`、`croak`和`confess`子例程，我们将在这一节后面的代码中使用这些子例程。与往常一样，可以查看在线文档了解有关Carp模块的更多信息。

这个类定义了以下方法：

#### CLASSNAME->TIESCALAR(*LIST*)

每次绑定一个标量变量时都会触发这个类的TIESCALAR方法。*LIST*可选，包含正确初始化对象所需的所有参数（在我们的例子中，TIESCALAR只有一个参数：即文件名）。这个方法应当返回一个对象，不过它并不一定是某个标量的引用。但在我们的例子中这确实是一个标量引用：

```
sub TIESCALAR {           # ScalarFile.pm中
    my $class = shift;
    my $filename = shift;
    $count++;             # 一个文件作用域词法变量，这是类的私有属性
    return bless \$filename, $class;
}
```

由于没有与匿名数组和散列生成器（`[]`和`{}`）等价的标量，我们只是用`bless`祝福一个词法作用域变量的指示对象，一旦这个名字出了作用域，实际上就会变为匿名。只要变量确实是词法作用域变量，这样就是可行的（可以对数组和散列做同样的处理）。如果尝试对一个全局变量使用这种技巧，你可能以为能成功，但创建另一个`camel.lot`时就会发现你的想法是错的。不要试图写类似这样的代码：

```
sub TIESCALAR { bless \$_[1], $_[0] }    # 不正确，
                                           # 可能指示全局变量。
```

如果编写一个更健壮的构造函数，可能要检查这个文件名是否可访问。首先查看这个文件是否可读，因为我们不希望破坏已有的值（换句话说，不能假设用户打算先写文件。他可能很珍惜上一次运行程序得到的老的Camel Lot文件）。如果不能打开或创建指定的文件名，我们会返回`undef`礼貌地提示错误，还可以（可选）通过`carp`

输出一个警告（也可以使用croak，这完全取决于你的个人偏好）。我们使用warnings pragma来确定用户对我们的警告是否感兴趣：

```
sub TIESCALAR { # 在ScalarFile.pm中
    my $class = shift;
    my $filename = shift;
    my $fh;
    if ( -r -w $filename ) {
        close $fh;
        $count++;
        return bless \$filename, $class;
    }
    carp "Can't tie $filename: $!" if warnings::enabled();
    return;
}
```

有了这样一个构造函数，现在可以关联标量\$string和文件camel.lot：

```
tie ($string, "ScalarFile", "camel.lot") || die;
```

我们还是做了一些本不该做的假设。在这个类的生产版本中，我们可能只打开一次文件句柄，并在绑定期间记住文件句柄以及文件名，用flock保证句柄在全程中是排他锁定的。否则，我们会面对竞争，参见第20章“处理计时问题”一节。

#### SELF->FETCH

只要访问绑定变量（也就是说，读取绑定变量的值），就会调用这个方法。除了绑定到变量的对象外，这个方法没有任何其他参数。在我们的例子中，这个对象包含文件名。

```
sub FETCH {
    my $self = shift;
    confess "I am not a class method" unless ref $self;
    return unless open my $fh, $$self;
    read($fh, my $value, -s $fh); # 注意：不要在管道上使用-s!
    return $value;
}
```

这一次我们决定当FETCH得到的不是一个引用时要“发点脾气”（产生一个异常）。可能会把它当作一个类方法来调用，或者有人可能错误地把它当作一个子例程来调用。我们没有其他方法返回错误，所以这可能是最合适的做法。实际上，一旦试图对\$self解引用，Perl就会产生一个异常；只是我们更有礼貌一些，而且还使用confess在用户屏幕上输出了一个完整的栈回溯记录（如果可以认为这是一种礼貌做法的话）。

现在如果有以下代码，就能看到camel.lot的内容了：

```
tie($string, "ScalarFile", "camel.lot");
print $string;
```

## SELF->STORE(VALUE)

设置绑定变量时（即赋值）会运行这个方法。第一个参数`SELF`是与变量关联的对象；`VALUE`是为这个变量赋的值（这里使用“赋值”一词并不严格，修改变量的任何操作都可以调用`STORE`）。

```
sub STORE {
    my($self,$value) = @_;
    ref($self)          || confess "not a class method";
    open(my $fh, ">", $$self) || croak "can't clobber $$self: $!";
    syswrite($fh, $value) == length $value
                          || croak "can't write to $$self: $!";
    close($fh)           || croak "can't close $$self: $!";
    return $value;
}
```

“赋值”之后，我们要返回这个新值，因为这正是赋值要做的事情。如果赋值未成功，则用`croak`显示这个错误。可能的原因也许是我们没有权限写文件，或者磁盘已满，也可能是磁盘控制器被弄坏了。有时你能控制魔法，但有时是魔法控制你。

现在可以用以下代码写入文件`camel.lot`：

```
tie($string, "ScalarFile", "camel.lot");
$string = "Here is the first line of camel.lot\n";
$string .= "And here is another line, automatically appended.\n";
```

## SELF->UNTIE

这个方法由`untie`触发，而且只能由`untie`触发。在这个例子中，它没有太大用处，所以这里只是指示调用了这个方法：

```
sub UNTIE {
    my $self = shift;
    confess "Untying!";
}
```

参见本章后面“一个解除绑定小陷阱”一节。

## SELF->DESTROY

与绑定变量关联的对象将被垃圾回收时会触发这个方法，以便对象做些特殊处理来完成清理工作。其他类很少需要这样一个方法，因为Perl会为你自动地释放这些“垂死”对象的内存。这里我们定义了一个`DESTROY`方法来递减绑定文件计数：

```
sub DESTROY {
    my $self = shift;
    confess "This is not a class method!" unless ref $self;
    $count--;
}
```

还可以提供一个额外的类方法获取当前计数。实际上，它并不关心作为一个类方法还是作为一个对象方法来调用，不过调用`DESTROY`之后，现在已经没有这个对象了，不是吗？



```

sub count {
    ### my $invocant = shift;
    $count;
}

```

任何时候都可以将它作为一个类方法来调用，如下：

```

if (ScalarFile->count) {
    warn "Still some tied ScalarFiles sitting around somewhere...\n";
}

```

这就是绑定标量的全部了。实际上，还可以比这更简单，因为这里为了做到完整性、健壮性和艺术性，我们还额外做了一些工作。你当然可以建立更简单的TIESCALAR类。

## 魔法计数器变量

这里给出一个简单的Tie::Counter类，这是受CPAN中同名模块的启发。绑定到这个类的变量每次使用时会自增1。例如：

```

tie my $counter, "Tie::Counter", 100;
@array = qw /Red Green Blue/;
for my $color (@array) { # 打印:
    print "$counter $color\n";
}

```

# 100 Red  
# 101 Green  
# 102 Blue

这个构造函数有一个可选的参数，即计数器的第一个值，这个值默认为0。为这个计数器赋值会设置一个新值。下面给出这个类：

```

package Tie::Counter;
sub FETCH      { ++ ${ $_[0] } }
sub STORE      { ${ $_[0] } = $_[1] }
sub TIESCALAR {
    my ($class, $value) = @_;
    $value = 0 unless defined $value;
    bless \$value => $class;
}
1; # 如果在模块中

```

看到了吧？这个类非常短小。建立这样一个类根本不需要多少代码。

## 循环处理值

借助tie的魔法，数组也可以被当作标量。tie接口可以把标量接口转换为数组接口。Tie::Cycle CPAN模块使用一个标量循环处理一个数组中的各个值。这个对象跟踪一个游标，每次访问时向前推进一步。到达末尾时，再回到最开始的位置：

```

package Tie::Cycle;

sub TIESCALAR {

```

```

    my $class = shift;
    my $list_ref = shift;
    return unless ref $list_ref eq ref [];
    my @shallow_copy = map { $_ } @$list_ref;
    my $self = [ 0, scalar @shallow_copy, \@shallow_copy ];
    bless $self, $class;
}

sub FETCH {
    my $self = shift;
    my $index = $$self[0]++;
    $$self[0] %= $self->[2];
    return $self->[2]->[ $index ];
}

sub STORE {
    my $self = shift;
    my $list_ref = shift;
    return unless ref $list_ref eq ref [];
    $self = [ 0, scalar @$list_ref, $list_ref ];
}

```

这样可以很方便地提供不同的CSS类，从而交替显示HTML表格中的行而不会让代码变得复杂：

```

tie my $row_class, "Tie::Cycle", [ qw(odd even) ];

for my $item (@items) {
    print qq(<tr class="$row_class">...</tr>);
}

```

采用这种方法，不用修改代码就能很容易地增加更多CSS类：

```

tie my $row_class, "Tie::Cycle", [ qw(red green blue) ];

```

## 魔法消除\$\_

这个奇特的underscore绑定类<sup>注2</sup>用来防止\$\_的非局部使用。并不是用use加载模块(use会调用类的import方法)，这个模块要用no加载，来调用很少使用的unimport方法（参见第11章）。假设有以下代码：

```

no underscore;

```

这样一来，如果将\$\_用作一个非局部的全局变量，就会产生一个异常。

下面是这个模块的一个小测试用例：

```

#!/usr/bin/perl
no underscore;

```

---

注2：说来奇怪，这个underscore来自本书之前版本中的一个例子，后来在《Perl Cookbook》中也有用到，Dan Kogai受它启发创建了一个CPAN模块。

```

@tests = (
    "Assignment" => sub { $_ = "Bad" },
    "Reading"    => sub { print },
    "Matching"   => sub { $x = /badness/ },
    "Chop"       => sub { chop },
    "Filetest"   => sub { -x },
    "Nesting"    => sub { for (1..3) { print } },
);

while ( ($name, $code) = splice(@tests, 0, 2) ) {
    print "Testing $name: ";
    eval { &$code };
    print "$@" ? "detected" : " missed!";
    print "\n";
}

```

这会打印以下结果：

```

Testing Assignment: detected
Testing Reading: detected
Testing Matching: detected
Testing Chop: detected
Testing Filetest: detected
Testing Nesting: 123 missed!

```

最后一个“missed”，这是因为它已由for循环正确地局部化，因此可以安全访问。

下面给出这个奇特的underscore模块（我们提到过吧？它很奇特）。它能正常工作是因为绑定魔法被local有效地隐藏了。这个模块在它自己的初始化代码中完成tie，所以也可以使用require：

```

package underscore;
use warnings;
use strict;
use Carp ();
our $VERSION = sprintf "%d.%02d", q$Revision: 0.1 $ =~ /(\d+)/g;

sub TIESCALAR{
    my ($pkg, $code, $msg) = @_;
    bless [$code, $msg], $pkg;
}

sub unimport {
    my $pkg = shift;
    my $action = shift;
    no strict "refs";
    my $code = ref $action
        ? $action
        : ($action
            ? \&{ "Carp::" . $action }
            : \&Carp::croak
        );
    my $msg = shift || '$_ is forbidden';
    untie $_ if tied $_;
}

```



```

    tie $_, _PACKAGE_, $code, $msg;
}

sub import{ untie $_ }

sub FETCH{ $_[0]->[0]($_[0]->[1]) }
sub STORE{ $_[0]->[0]($_[0]->[1]) }

1; # underscore结束

```

在程序中很难对这个类混合使用`use`和`no`调用，因为它们都发生在编译时，而不是运行时。类似于`use`和`no`，可以直接调用`Underscore->import`和`Underscore->unimport`。不过，正常情况下，要想“反悔”，希望重新自由地使用`$_`，只需直接使用`local`，这就是关键。

## 绑定数组

实现一个绑定数组的类必须至少定义`TIEARRAY`、`FETCH`和`STORE`方法。另外还有很多可选的方法：当然包括无处不在的`UNTIE`和`DESTROY`方法，以及用来提供`$#array`和`scalar(@array)`访问的`STORESIZE`和`FETCHSIZE`方法。另外，Perl需要清空数组时会触发`CLEAR`方法，Perl为一个实际数组预先扩展分配空间时会触发`EXTEND`方法。

如果还希望能够在绑定数组上使用相应的Perl函数，如`POP`、`PUSH`、`SHIFT`、`UNSHIFT`、`SPLICE`、`DELETE`和`EXISTS`方法，也可以定义这些方法。可以将`Tie::Array`类作为基类，使用`FETCH`和`STORE`实现其中前5个函数（`Tie::Array`的`DELETE`和`EXISTS`的默认实现只是调用`croak`）。只要定义了`FETCH`和`STORE`，对象包含何种类型的数据结构并不重要。

另一方面，`Tie::StdArray`类（在标准的`Tie::Array`模块中定义）提供了一个基类，其中包含一些默认方法，这些方法假设对象包含一个常规数组。下面是一个简单的数组绑定类，这里就使用了这个类作为基类。由于它使用`Tie::StdArray`作为基类，所以只需要定义需要以非标准方式处理的方法：

```

#!/usr/bin/perl
package ClockArray;
use Tie::Array;
our @ISA = "Tie::StdArray";
sub FETCH {
    my($self,$place) = @_;
    $self->[ $place % 12 ];
}
sub STORE {
    my($self,$place,$value) = @_;
    $self->[ $place % 12 ] = $value;
}

package main;
tie my @array, "ClockArray";

```

```
@array = ( "a" ... "z" );
print "@array\n";
```

运行时，程序会打印结果“y z o p q r s t u v w x”。这个类提供了一个只有12个槽的数组，就像时钟上的小时一样，编号为0到11。如果请求第15个数组索引，实际上会得到第3个索引。可以把它看作是一个旅行助手，能帮助那些不会读24小时制时钟的人。

## 数组绑定方法

前面介绍的内容很简单。下面来看内部细节。为了便于说明，我们会实现一个数组，这个数组的上下界在创建时确定。如果想访问越界的内容，就会产生一个异常。例如：

```
use BoundedArray;
tie @array, "BoundedArray", 2;

$array[0] = "fine";
$array[1] = "good";
$array[2] = "great";
$array[3] = "whoa"; # 禁止；这会显示一个错误消息。
```

这个类的声明代码如下：

```
package BoundedArray;
use Carp;
use strict;
```

为了避免以后定义SPLICE，我们将继承Tie::Array类：

```
use Tie::Array;
our @ISA = ("Tie::Array");
```

### CLASSNAME->TIEARRAY(LIST)

作为这个类的构造函数，TIEARRAY应当返回一个被祝福的引用，通过这个引用可以模拟绑定数组。

下一个例子只是为了说明实际上不一定非得返回一个数组引用，这里选择了一个散列引用来表示我们的对象。散列很适合作为一个通用的记录类型：散列的“BOUND”键对应的值将存储所允许的上界，其“DATA”值包含实际数据。如果试图在这个类以外解引用所返回的对象（不加怀疑地认为它是一个数组引用），就会产生一个异常。

```
sub TIEARRAY {
    my $class = shift;
    my $bound = shift;
    confess "usage: tie(\@ary, 'BoundedArray', max_subscript)"
        if @_ || $bound =~ /\D/;
    return bless { BOUND => $bound, DATA => [] }, $class;
}
```

现在可以写为：

```
tie(@array, "BoundedArray", 3); # 允许的最大索引值为3
```

以确保这个数组不会超过4个元素。只要访问或存储数组的一个元素，就会像标量一样调用FETCH和STORE，不过有一个额外的索引参数。

#### SELF->FETCH(INDEX)

只要访问绑定数组中的一个元素就会运行这个方法。在对象参数后面，这个方法还需要另外一个参数：想要获取的值的相应索引。

```
sub FETCH {  
    my ($self, $index) = @_;  
    if ($index > $self->{BOUND}) {  
        confess "Array OOB: $index > $self->{BOUND}";  
    }  
    return $self->{DATA}[$index];  
}
```

#### SELF->STORE(INDEX, VALUE)

需要设置绑定数组中的一个元素时就会调用这个方法。在对象参数后面，这个方法还需要两个参数：要在哪个索引上存储值，以及在这个位置上存放什么值。例如：

```
sub STORE {  
    my ($self, $index, $value) = @_;  
    if ($index > $self->{BOUND}) {  
        confess "Array OOB: $index > $self->{BOUND}";  
    }  
    return $self->{DATA}[$index] = $value;  
}
```

#### SELF->UNTIE

这个方法由untie触发。这个例子中并不需要这个方法。有关的注意事项见这一章后面“一个解除绑定小陷阱”一节。

#### SELF->DESTROY

需要撤销绑定变量并回收其内存时，Perl会调用这个方法。提供了垃圾回收的语言中几乎都不需要这个方法，所以这个例子没有使用这个方法。

#### SELF->FETCHSIZE

FETCHSIZE方法应当返回与SELF关联的绑定数组中的元素总数。它等价于scalar(@array)，这通常等于\$#array + 1。

```
sub FETCHSIZE {  
    my $self = shift;  
    return scalar @{$self->{DATA}};  
}
```

#### SELF->STORESIZE(COUNT)

这个方法将与SELF关联的绑定数组中的元素总数设置为COUNT。如果数组收缩，应当



删除超出`COUNT`的元素。如果数组扩展，应当确保新位置（新元素）未定义。对于我们的`BoundedArray`类，还要确保数组扩展不会超出最初设置的限制。

```
sub STORESIZE {
    my ($self, $count) = @_;
    if ($count > $self->{BOUND}) {
        confess "Array OOB: $count > $self->{BOUND}";
    }
    ${$self->{DATA}} = $count;
}
```

#### `SELF->EXTEND(COUNT)`

Perl使用`EXTEND`方法来指示数组可能要扩展以便保存`COUNT`个元素。这样可以一次分配足够大的内存，而不是通过多个连续的调用来分配。由于`BoundedArrays`有固定的上界，所以我们没有定义这个方法。

#### `SELF->EXISTS(INDEX)`

这个方法验证绑定数组中`INDEX`对应的元素是否存在。对于我们的`BoundedArray`，在验证未超过固定上界之后，我们直接使用了Perl的内置`exists`。

```
sub EXISTS {
    my ($self, $index) = @_;
    if ($index > $self->{BOUND}) {
        confess "Array OOB: $index > $self->{BOUND}";
    }
    exists ${$self->{DATA}}[$index];
}
```

#### `SELF->DELETE(INDEX)`

`DELETE`方法删除绑定数组`SELF`中`INDEX`对应的元素。对于我们的`BoundedArray`类，这个方法看上去与`EXISTS`几乎完全相同，不过这不是标准。

```
sub DELETE {
    my ($self, $index) = @_;
    print STDERR "deleting!\n";
    if ($index > $self->{BOUND}) {
        confess "Array OOB: $index > $self->{BOUND}";
    }
    delete ${$self->{DATA}}[$index];
}
```

#### `SELF->CLEAR`

清空数组时会调用这个方法。将数组设置为一个新值列表（或一个空列表）时就会发生这种情况，不过提供给`undef`函数时不会清空。由于清空的`BoundedArray`总能满足上界限制，所以这里不需要做任何检查：

```
sub CLEAR {
    my $self = shift;
    ${$self->{DATA}} = [];
}
```

如果把数组设置为一个列表，会触发CLEAR，但是不会看到列表值。所以如果超过了上界，如下：

```
tie(@array, "BoundedArray", 2);  
@array = (1, 2, 3, 4);
```

CLEAR方法仍能成功地返回。只会在随后的STORE中产生异常。这个赋值会触发一个CLEAR和4个STORE。

#### SELF->PUSH(*LIST*)

这个方法把*LIST*的元素追加到数组末尾。对于我们的BoundedArray类，这个方法大致如下：

```
sub PUSH {  
    my $self = shift;  
    if (@_ + ${$self->{DATA}} > $self->{BOUND}) {  
        confess "Attempt to push too many elements";  
    }  
    push @{$self->{DATA}}, @_  
}
```

#### SELF->UNSHIFT(*LIST*)

这个方法把*LIST*的元素追加到数组开头。对于我们的BoundedArrayc类，这个子例程类似于PUSH。

#### SELF->POP

POP方法删除数组的最后一个元素，并将其返回。对于BoundedArray，这个方法只有一行代码：

```
sub POP { my $self = shift; pop @{$self->{DATA}} }
```

#### SELF->SHIFT

SHIFT方法删除数组中的第一个元素，并将其返回。对于BoundedArray，这个方法类似于POP。

#### SELF->SPLICE(*OFFSET*, *LENGTH*, *LIST*)

这个方法允许对SELF数组分片。为了模拟Perl的内置splice函数，*OFFSET*应当是可选的，默认为0，负值表示从数组末尾倒数。*LENGTH*也应当是可选的，默认为数组的剩余长度。*LIST*可以为空。如果能正确地模拟Perl的内置splice函数，这个方法会返回原来位于*OFFSET*的*LENGTH*个元素的一个列表（也就是说，要被*LIST*替换的元素列表）。

由于分片操作有些复杂，我们不打算定义这个方法。这里只使用Tie::Array模块的SPLICE子例程，从Tie::Array继承时就可以得到这个子例程。这样一来，我们可以用其他BoundedArray方法定义SPLICE，还能完成越界检查。

以上就是整个BoundedArray类。它稍稍改变了数组的语义。不过我们还可以做得更好，而且需要更小的空间。

## 概念上的便利

关于变量很好的一点是它们可以内插。而函数不好的一点就是它们不能内插。可以使用绑定数组建立一个可以内插的函数。假设你想将随机整数内插到一个字符串中。可以写以下代码：

```
#!/usr/bin/perl
package RandInterp;
sub TIEARRAY { bless \my $self };
sub FETCH { int rand $_[1] };

package main;
tie @rand, "RandInterp";
for (1,10,100,1000) {
    print "A random integer less than $_ would be $rand[$_]\n";
}
$rand[32] = 5; # 这会重新格式化我们的系统盘吗？
```

运行时，会打印以下结果：

```
A random integer less than 1 would be 0
A random integer less than 10 would be 3
A random integer less than 100 would be 46
A random integer less than 1000 would be 755
Can't locate object method "STORE" via package "RandInterp" at foo line 10.
```

可以看到，没有实现STORE并没有大问题，只是与往常一样会提示错误。

## 绑定散列

实现绑定散列的类应当定义8个方法。TIEHASH构造新对象。FETCH和STORE访问键/值对。EXISTS报告一个键在散列中是否存在，DELETE删除一个键及其关联的值<sup>注3</sup>。CLEAR删除所有键/值对，将散列清空。调用keys、values或each时，FIRSTKEY和NEXTKEY会迭代处理键/值对。另外，与以往一样，如果希望在撤销对象时完成一些特定的动作，可以定义一个DESTROY方法（如果认为看起来方法很多，这说明你肯定没有认真阅读以上关于数组的各个小节。不管怎样，完全可以从标准Tie::Hash模块继承默认方法，只需要重新定义那些有意思的方法。另外，Tie::StdHash假设其实现也是一个散列）。

例如，假设你想创建一个散列，每次为一个键赋值时，并不是覆盖之前的内容，而是将新值追加到值数组中。这样一来，如果有以下代码：

---

注3： 要记住，Perl会区别散列中不存在的键和散列中存在但相应值为undef的键。可以分别用exists和defined来测试这两种可能性。



```
$h{$k} = "one";
$h{$k} = "two";
```

实际的工作是：

```
push @{ $h{$k} }, "one";
push @{ $h{$k} }, "two";
```

这并不是一个很复杂的概念，所以应该能用一个很简单的模块实现。可以使用 Tie::StdHash 作为基类（真是这样的）。下面给出一个 Tie::AppendHash，它就是这样做的：

```
package Tie::AppendHash;
use Tie::Hash;
our @ISA = ("Tie::StdHash");
sub STORE {
    my ($self, $key, $value) = @_;
    push @{$self->{key}}, $value;
}
1;
```

## 散列绑定方法

这里给出一个很有意思的绑定散列类的例子：它会提供一个散列，表示一个特定用户的点文件（所谓点文件就是名字以一个点号开头的文件，这是 Unix 下表示初始化文件的一个命名约定）。可以用文件名作为散列中的键（去掉点号），以此得到点文件的内容。例如：

```
use DotFiles;
tie %dot, "DotFiles";
if ( $dot{profile} =~ /MANPATH/ ||
     $dot{login} =~ /MANPATH/ ||
     $dot{cshrc} =~ /MANPATH/ ) {
    print "you seem to set your MANPATH\n";
}
```

下面是使用这个绑定类的另一种方法：

```
# 第3个参数是用户名（我们要绑定到该用户的点文件）
tie %him, "DotFiles", "daemon";
foreach $f (keys %him) {
    printf "daemon dot file %s is size %d\n", $f, length $him{$f};
}
```

在 DotFiles 例子中，我们将对象实现为一个常规的散列，其中包含几个重要的字段，在这些字段中，只有 {CONTENTS} 字段包含用户所认为的散列。表 14-1 给出了这个对象的具体字段。

表14-1: DotFiles中的对象字段

字段	内容
USER	这个对象表示哪个用户的点文件
HOME	这些点文件所在的位置
CLOBBER	是否允许修改或删除这些点文件
CONTENTS	具体的散列，包含点文件名与内容的映射

下面是*DotFiles.pm*的开始部分（声明代码）：

```
package DotFiles;
use Carp;
sub whowasi { (caller(1))[3] . "()" }
my $DEBUG = 0;
sub debug { $DEBUG = @_ ? shift : 1 }
```

对于这个例子，我们希望能够打开调试输出，以便跟踪开发过程，所以我们建立了\$DEBUG。另外我们还在内部建立了一个便利函数来帮助打印警告消息：whowasi会返回调用当前函数的那个函数的名字（whowasi的“祖父”函数）。

下面是DotFiles绑定散列的方法：

CLASSNAME->TIEHASH(LIST)

以下是DotFiles构造函数：

```
sub TIEHASH {
    my $self = shift;
    my $user = shift || $>;
    my $dotdir = shift || "";

    croak "usage: @{$[ &whowasi ]} [USER [DOTDIR]]" if @_;

    $user = getpwuid($user) if $user =~ /^\\d+$/;
    my $dir = (getpwnam($user))[7]
        || croak "@{[ &whowasi ]}: no user $user";
    $dir .= "/$dotdir" if $dotdir;

    my $node = {
        USER          => $user,
        HOME           => $dir,
        CONTENTS       => {},
        CLOBBER        => 0,
    };

    opendir(DIR, $dir)
        || croak "@{[ &whowasi ]}: can't opendir $dir: $!";
    for my $dot (grep /^\\. / && -f "$dir/$_", readdir(DIR)) {
        $dot =~ s/^\\. //;
        $node->{CONTENTS}{$dot} = undef;
    }
}
```

```

        closedir DIR;

        return bless $node, $self;
    }

```

值得一提的是，如果你打算对以上`readdir`返回的值应用文件测试，最好在文件名前面追加当前目录（就像我们所做的一样）。否则，由于没有使用`chdir`切换目录，你测试的很可能并不是你原本想要测试的文件。

### SELF->FETCH(KEY)

这个方法读取绑定散列中的一个元素。在对象参数之后，该方法还有一个参数：想要获取的值对应的键。这个键是一个字符串，可以对它做你想做的任何处理（即任何字符串操作）。

下面给出`DotFiles`类的获取方法：

```

sub FETCH {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $dir = $self->{HOME};
    my $file = "$dir/.$dot";

    unless (exists $self->{CONTENTS}->{$dot} || -f $file) {
        carp "@{&whowasi}: no $dot file" if $DEBUG;
        return undef;
    }

    # 实现一个缓存
    if (defined $self->{CONTENTS}->{$dot}) {
        return $self->{CONTENTS}->{$dot};
    } else {
        return $self->{CONTENTS}->{$dot} = `cat $dir/.$dot`;
    }
}

```

这里我们用了一个小技巧，运行了Unix `cat(1)`命令，不过更可移植的做法（也更高效）是我们自己打开文件。另一方面，由于点文件是一个Unix概念，所以我们不太担心这个问题。或者说，也不应该担心。

### SELF->STORE(KEY, VALUE)

在绑定散列中设置（写）一个元素时，这个`STORE`方法会完成具体的工作。在对象参数后面还有两个参数：存储新值所对应的键，以及要存储的值本身。

对于我们的`DotFiles`例子，如果没有首先在`tie`返回的原始对象上调用`clobber`方法，我们不允许用户覆盖文件：

```

sub STORE {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;

```



```

my $value = shift;
my $file = $self->{HOME} . "/.$dot";

croak "@{&whowasi}: $file not clobberable"
    unless $self->{CLOBBER};
open(F, "> $file") || croak "can't open $file: $!";
print F $value;
close(F) || croak "can't close $file: $!";
}

```

如果有人想做些修改，可能会写以下代码：

```

$obj = tie %daemon_dots, "daemon";
$obj->clobber(1);
$daemon_dots{signature} = "A true daemon\n";

```

不过，完全可以用`tied`设置`{CLOBBER}`：

```

tie %daemon_dots, "DotFiles", "daemon";
tied(%daemon_dots)->clobber(1);

```

或者写为一条语句：

```

(tie %daemon_dots, "DotFiles", "daemon")->clobber(1);

```

`clobber`方法很简单：

```

sub clobber {
    my $self = shift;
    $self->{CLOBBER} = @_ ? shift : 1;
}

```

## SELF->DELETE(KEY)

这个方法处理从散列中删除一个元素的请求。如果你模拟的散列使用了某个地方的一个真正的散列，可以直接调用真正的`delete`。另外，要当心检查用户是否真的想要删除文件：

```

sub DELETE {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $file = $self->{HOME} . "/.$dot";
    croak "@{&whowasi}: won't remove file $file"
        unless $self->{CLOBBER};
    delete $self->{CONTENTS}->{$dot};
    unlink($file) || carp "@{&whowasi}: can't unlink $file: $!";
}

```

## SELF->CLEAR

需要清空整个散列时会运行这个方法，通常是为其赋一个空列表。在我们的例子中，这会删除用户的所有点文件！这样做的危险性极大，所以我们要求做这个操作之前`CLOBBER`必须设置为大于1：

```

sub CLEAR {

```

```

    carp &whowasi if $DEBUG;
    my $self = shift;
    croak "@{[&whowasi]}: won't remove all dotfiles for $self->{USER}"
        unless $self->{CLOBBER} > 1;
    for my $dot ( keys %{$self->{CONTENTS}} ) {
        $self->DELETE($dot);
    }
}

```

### SELF->EXISTS(KEY)

用户对一个特定的散列调用`exists`函数时就会运行这个方法。在我们的例子中，会查看`{CONTENTS}`散列元素来找出答案：

```

sub EXISTS {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    return exists $self->{CONTENTS}->{$dot};
}

```

### SELF->FIRSTKEY

用户开始迭代处理散列（如通过一个`keys`、`values`或`each`调用）时会调用这个方法。通过在标量上下文中调用`keys`，可以重置其内部状态，确保`return`语句中使用的下一个`each`会得到第一个键。

```

sub FIRSTKEY {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $temp = keys %{$self->{CONTENTS}};
    return scalar each %{$self->{CONTENTS}};
}

```

### SELF->NEXTKEY(PREVKEY)

这个方法是`keys`、`values`或`each`函数的迭代器。`PREVKEY`是最后访问的键，Perl知道如何提供这个键。如果`NEXTKEY`方法需要知道它的前一个状态来计算下一个状态，这个方法就很有用。

对于我们的例子，要使用一个真正的散列来表示绑定散列的数据，只不过这个散列存储在该散列的`CONTENTS`字段中而不是存储在散列本身。所以可以利用Perl的`each`迭代器：

```

sub NEXTKEY {
    carp &whowasi if $DEBUG;
    my $self = shift;
    return scalar each %{$self->{CONTENTS}}
}

```

### SELF->UNTIE

这个方法由`untie`触发。这个例子中并不需要这个方法。有关的注意事项见这一章后面“一个解除绑定小陷阱”一节。

## SELF->DESTROY

需要撤销一个绑定散列的对象时就会触发这个方法。除了用于调试和额外的清理工作外，实际上一般并不需要这个方法。下面是一个非常简单的版本：

```
sub DESTROY {  
    carp &whowasi if $DEBUG;  
}
```

以上我们已经给出了所有方法，你的任务就是退回去找出哪里内插了@{&whowasi}，并把它们替换为一个简单的绑定标量\$whowasi完成相同的工作。

## 绑定文件句柄

实现绑定文件句柄的类应当定义以下方法：TIEHANDLE以及PRINT、PRINTF、WRITE、READLINE、GETC和READ中至少一个方法。这个类还可以提供一个DESTROY方法，以及BINMODE、OPEN、CLOSE、EOF、FILENO、SEEK、TELL、READ和WRITE方法，从而可以对绑定文件句柄完成相应的Perl内置操作（嗯，这并不完全正确：WRITE对应syswrite，它与Perl的内置write函数没有任何关系，内置write函数要结合format声明完成打印）。

Perl嵌入在另一个程序中时（如Apache或vi），或者提供给STDOUT或STDERR的输出需要以某种特殊的方式重定向时，绑定文件句柄尤其有用。

不过文件句柄实际上并不一定非得绑定到一个文件。可以使用输出语句建立一个内存中数据结构，并使用输入语句读回有关内容。下面可以很容易地反转print和printf语句的打印顺序，但不用反转打印代码：

```
package ReversePrint 0.01 {  
    use strict;  
    sub TIEHANDLE {  
        my $class = shift;  
        bless [], $class;  
    }  
    sub PRINT {  
        my $self = shift;  
        push @$self, join("", => @_);  
    }  
    sub PRINTF {  
        my $self = shift;  
        my $fmt = shift;  
        push @$self, sprintf($fmt, @_);  
    }  
    sub READLINE {  
        my $self = shift;  
        pop @$self;  
    }  
}  
  
my $m = "--MORE--\n";
```



```

tie *REV, "ReversePrint";

# 做一些prints和printfs调用
print REV "The fox is now dead.$m";

printf REV <<"END", int rand 10000000;
The quick brown fox jumps
over the lazy dog %d times!
END

print REV <<"END";
The quick brown fox jumps
over the lazy dog.
END

# 现在从同一个句柄读回
print while <REV>;

```

这会打印以下结果：

```

The quick brown fox jumps
over the lazy dog.
The quick brown fox jumps
over the lazy dog 3179357 times!
The fox is now dead.--MORE--

```

## 文件句柄绑定方法

对于我们的扩展例子，这里将创建一个文件句柄，它把打印到该文件的字符串转换为大写。打开文件时开头打印<SHOUT>，关闭时末尾打印</SHOUT>，这只是为了更有趣一点。这样我们就能使用良构的XML了。

以下是*Shout.pm*文件的开始部分，将由该文件实现这个类：

```

package Shout;
use Carp;          # 以便汇报错误

```

下面列出*Shout.pm*中的方法定义。

### CLASSNAME->TIEHANDLE(*LIST*)

这是这个类的构造函数，与往常一样，这个方法要返回一个被祝福的引用。

```

sub TIEHANDLE {
    my $class = shift;
    my $form = shift;
    open(my $self, $form, @_) || croak "can't open $form@_: $!";
    if ($form =~ />/) {
        print $self "<SHOUT>\n";
        $$self->{WRITING} = 1;    # 记得要写结束标记
    }
    return bless $self, $class;    # $self是一个类型团引用
}

```

这里我们根据传入`tie`操作符的模式和文件名打开一个新文件句柄，将`<SHOUT>`写入文件，然后返回指向这个文件的一个被祝福的引用。这个`open`语句里有很多内容，不过这里只强调一点，除了通常的“open or die”惯例外，`my $self`会为`open`提供一个未定义的标量，它知道如何将它自动生成为一个类型团。这是一个类型团的事实也很重要，因为类型团不仅包含文件的真正I/O对象，还包含各种其他方便的数据结构可以自由使用，如标量（`$$$self`）、数组（`@$self`）和散列（`%$self`）。我们没有提到子例程`&$self`。

`$form`是文件名或模式参数。如果它是一个文件名，则`@_`为空，所以这会表现得像一个两参数的`open`。否则，`$form`是其余参数的模式。

在`open`后面，要查看是否要写开始标记。如果是，则写入这个开始标记。接下来就要使用前面提到的那些类型团数据结构中的某一个数据结构。`$$self->{WRITING}`是使用类型团存储有趣信息的一个例子。在这种情况下，我们要记住是否写过开始标记，从而知道是否要写相应的结束标记。我们使用了`%$self`散列，这样可以为字段提供一个合适的名字。也可以使用标量`$$$self`，不过这样不具有自描述性，不能清楚地表明它的作用（或者它只能描述自己，这取决于你的看法）。

#### `SELF->PRINT(LIST)`

这个方法为绑定句柄实现`print`。`LIST`是传入`print`的所有内容。下面的方法会把`LIST`的各个元素变为大写：

```
sub PRINT {
    my $self = shift;
    print $self map {uc} @_;
}
```

#### `SELF->READLINE`

用尖角操作符（`<FH>`）或`readline`读取文件句柄时，这个`READLINE`方法会提供数据。如果没有更多数据，这个方法要返回`undef`。

```
sub READLINE {
    my $self = shift;
    return <$self>;
}
```

这里只是返回了`<$self>`，从而能根据在标量上下文还是列表上下文调用来提供适当地表现。

#### `SELF->GETC`

在绑定文件句柄上使用`getc`时就会运行这个`GETC`方法。

```
sub GETC {
    my $self = shift;
    return getc($self);
}
```

与Shout类中的很多其他方法一样，GETC方法只是调用相应的Perl内置函数，并返回结果。

### SELF->OPEN(LIST)

我们的TIEHANDLE方法本身会打开一个文件，使用Shout类的程序随后调用open时会触发这个方法。

```
sub OPEN {
    my $self = shift;
    my $form = shift;
    my $name = "$form@";
    $self->CLOSE;
    open($self, $form, @_) || croak "can't reopen $name: $!";
    if ($form =~ />/) {
        (print $self "<SHOUT>\n") || croak "can't start print: $!";
        $$self->{WRITING} = 1; # 记得要加结束标记
    }
    else {
        $$self->{WRITING} = 0; # 记得不要加结束标记
    }
    return 1;
}
```

我们调用自己的CLOSE方法来显式关闭文件，以防用户没有关闭文件。然后用open中指定的文件名打开一个新文件，并处理这个新文件。

### SELF->CLOSE

这个方法处理关闭句柄的请求。这里我们会搜索文件末尾，如果成功，则在使用Perl的内置close之前先打印</SHOUT>。

```
sub CLOSE {
    my $self = shift;
    if ($$self->{WRITING}) {
        $self->SEEK(0, 2) || return;
        $self->PRINT("</SHOUT>\n") || return;
    }
    return close $self;
}
```

### SELF->SEEK(LIST)

在一个绑定文件句柄上搜索时，会调用SEEK方法。

```
sub SEEK {
    my $self = shift;
    my ($offset, $whence) = @_;
    return seek($self, $offset, $whence);
}
```

### SELF->TELL

在绑定句柄上使用tell时会调用这个方法。

```
sub TELL {
```



```

    my $self = shift;
    return tell $self;
}

```

### SELF->PRINTF(*LIST*)

在绑定句柄上使用printf时会运行这个方法。*LIST*会包含格式和要打印的元素。

```

sub PRINTF {
    my $self = shift;
    my $template = shift;
    return $self->PRINT(sprintf $template, @_);
}

```

这里使用sprintf生成格式化字符串，再把它传入PRINT转换为大写。不过，不一定非得使用内置的sprintf函数，可以截获百分号转义来达到你的目的。

### SELF->READ(*LIST*)

使用read或sysread读取句柄时这个方法会做出响应。需要说明，我们要“原地”修改*LIST*的第一个参数，以模拟read的功能，填入传入的标量作为其第二个参数。

```

sub READ {
    my ($self, undef, $length, $offset) = @_;
    my $bufref = \$_[1];
    return read($self, $$bufref, $length, $offset);
}

```

### SELF->WRITE(*LIST*)

用syswrite写入句柄时会调用这个方法。这里我们将待写入的字符串转换为大写。

```

sub WRITE {
    my $self = shift;
    my $string = uc(shift);
    my $length = shift || length $string;
    my $offset = shift || 0;
    return syswrite $self, $string, $length, $offset;
}

```

### SELF->EOF

使用eof测试一个绑定到Shout类的文件句柄是否到达文件末尾时，这个方法会返回一个布尔值。

```

sub EOF {
    my $self = shift;
    return eof $self;
}

```

### SELF->BINMODE(*IOLAYER*)

这个方法指定文件句柄上使用的I/O层。如果没有指定，则为区分文本和二进制文件的文件系统将绑定文件句柄置于二进制模式(:raw层)。

```

sub BINMODE {
    my $self = shift;

```

```

        my $disc = shift || ":raw";
        return binmode $self, $disc;
    }

```

你可能会这样写代码，不过实际上我们的情况有所例外，因为open已经向句柄写入了内容。所以对于我们的情况，可能应当是这样：

```

sub BINMODE { croak("Too late to use binmode") }

```

## SELF->FILENO

这个方法要返回操作系统为绑定文件句柄关联的文件描述符（fileno）。

```

sub FILENO {
    my $self = shift;
    return fileno $self;
}

```

## SELF->UNTIE

这个方法由untie触发。这个例子中并不需要这个方法。有关的注意事项见这一章后面“一个解除绑定小陷阱”一节。

## SELF->DESTROY

与其他类型的绑定类似，绑定对象将要撤销时会触发这个方法。这对于允许对象自行清理很有用。这里我们要确保文件已经关闭，以免程序忘记调用close。可以直接调用close \$self，不过最好是调用这个类的CLOSE方法。这样一来，即使类的设计者决定改变关闭文件的方式，这个DESTROY方法也不用修改。

```

sub DESTROY {
    my $self = shift;
    $self->CLOSE; # 使用Shout的CLOSE方法关闭文件
}

```

下面是Shout类的一个应用演示：

```

#!/usr/bin/perl
use Shout;
tie(*F00, Shout::, ">filename");
print F00 "hello\n";
seek F00, 0, 0;
@lines = <F00>;
close F00;
open(F00, "+<", "filename");
seek(F00, 8, 0);
sysread(F00, $inbuf, 5);
print "found $inbuf\n";
seek(F00, -5, 1);
syswrite(F00, "ciao!\n", 6);
untie(*F00);

```

# 打印HELLO  
# 返回到开始位置  
# 调用READLINE方法  
# 显式地关闭文件  
# 重新打开F00，调用OPEN  
# 跳过"<SHOUT>\n"  
# 从F00读5个字节到\$inbuf中  
# 应打印"hello"  
# 退回到"hello"前面  
# 向F00写6个字节  
# 隐式调用CLOSE方法

运行这个代码后，文件将包含以下内容：

```

<SHOUT>

```

```
CIAO!  
</SHOUT>
```

下面对这个内部类型团做一些更奇怪、更有意思的处理。像从前一样，我们使用同一个散列，不过增加了新键PATHNAME和DEBUG。首先，安装一个字符串化重载，这样打印对象时就会显示路径名（参见第13章）：

```
# 太酷了!  
use overload q("") => sub { $_[0]->pathname };  
  
# 这是桩函数，可以在这里放入你想跟踪的各个函数。  
sub trace {  
    my $self = shift;  
    local $Carp::CarpLevel = 1;  
    Carp::cluck("\ntrace magical method") if $self->debug;  
}  
  
# 重载处理方法来打印路径。  
sub pathname {  
    my $self = shift;  
    confess "i am not a class method" unless ref $self;  
    $$self->{PATHNAME} = shift if @_;  
    return $$self->{PATHNAME};  
}  
  
# 双重模式。  
sub debug {  
    my $self = shift;  
    my $var = ref $self ? \$$self->{DEBUG} : \our $Debug;  
    $$var = shift if @_;  
    return ref $self ? $$self->{DEBUG} || $Debug : $Debug;  
}
```

然后可以在所有普通方法的入口调用trace，如下：

```
sub GETC { $_[0]->trace;          # 新加代码  
    my($self) = @_;  
    getc($self);  
}
```

另外在TIEHANDLE和OPEN中设置路径名：

```
sub TIEHANDLE {  
    my $class = shift;  
    my $form = shift;  
    my $name = "$form@_";          # NEW  
    open(my $self, $form, @_ ) || croak "can't open $name: $!";  
    if ($form =~ />/) {  
        print $self "<SHOUT>\n";  
        $$self->{WRITING} = 1;     # 记得要写结束标记  
    }  
    bless $self, $class;           # $fh是一个类型团引用  
    $self->pathname($name);        # 新加代码  
    return $self;  
}
```



```

sub OPEN { $_[0]->trace;                # 新加代码
    my $self = shift;
    my $form = shift;
    my $name = "$form@";
    $self->CLOSE;
    open($self, $form, @_) || croak "can't reopen $name: $!";
    $self->pathname($name); # 新加代码
    if ($form =~ />/) {
        (print $self "<SHOUT>\n") || croak "can't start print: $!";
        $$self->{WRITING} = 1; # 记得要写结束标记。
    }
    else {
        $$self->{WRITING} = 0; # 记得不要写结束标记。
    }
    return 1;
}

```

还要在某个地方调用\$self->debug(1)来打开调试。这样我们的所有Carp::cluck调用就会生成有意义的消息。完成以上reopen会得到以下消息。这里显示了3个深藏的方法调用，这是在关闭老文件准备打开新文件时调用的：

```

trace magical method at foo line 87
Shout::SEEK('>filename', '>filename', 0, 2) called at foo line 81
Shout::CLOSE('>filename') called at foo line 65
Shout::OPEN('>filename', '+<', 'filename') called at foo line 141

```

## 有创意的文件句柄

可以把同一个文件句柄同时绑定到一个两端管道的输入和输出。假设你想以这种方式运行bc(1)（任意精度计算器）程序：

```

use Tie::Open2;

tie *CALC, "Tie::Open2", "bc -l";
$sum = 2;
for (1 .. 7) {
    print CALC "$sum * $sum\n";
    $sum = <CALC>;
    print "$_: $sum";
    chomp $sum;
}
close CALC;

```

可能希望它打印以下结果：

```

1: 4
2: 16
3: 256
4: 65536
5: 4294967296
6: 18446744073709551616
7: 340282366920938463463374607431768211456

```

如果你的计算机上有`bc(1)`程序，而且已经定义了以下的`Tie::Open2`，那么确实能看到上面预期的输出。这一次我们使用一个已祝福的数组来表示内部对象。它包含两个具体的文件句柄，分别用于读写（打开双端管道的具体工作由`IPC::Open2`完成，我们只做有意思的部分）。

```
package Tie::Open2;
use strict;
use Carp;
use Tie::Handle; # 不要继承这个类!
use IPC::Open2;

sub TIEHANDLE {
    my ($class, @cmd) = @_;
    no warnings "once";
    my @fhpair = \do { local(*RDR, *WTR) };
    bless $_, "Tie::StdHandle" for @fhpair;
    bless(\@fhpair => $class)->OPEN(@cmd) || die;
    return \@fhpair;
}

sub OPEN {
    my ($self, @cmd) = @_;
    $self->CLOSE if grep {defined} @{$self->FILENO };
    open2(@$self, @cmd);
}

sub FILENO {
    my $self = shift;
    [ map { fileno $self->[$_] } 0,1 ];
}

for my $outmeth ( qw(PRINT PRINTF WRITE) ) {
    no strict "refs";
    *$outmeth = sub {
        my $self = shift;
        $self->[1]->$outmeth(@_);
    };
}

for my $inmeth ( qw(READ READLINE GETC) ) {
    no strict "refs";
    *$inmeth = sub {
        my $self = shift;
        $self->[0]->$inmeth(@_);
    };
}

for my $doppelmeth ( qw(BINMODE CLOSE EOF)) {
    no strict "refs";
    *$doppelmeth = sub {
        my $self = shift;
        $self->[0]->$doppelmeth(@_) && $self->[1]->$doppelmeth(@_);
    };
}

for my $deadmeth ( qw(SEEK TELL)) {
    no strict "refs";
```

```

        *$deadmeth = sub {
            croak("can't $deadmeth a pipe");
        };
    }
    1;

```

最后4个循环在我们看来确实很新奇。要了解这里在做什么，可以查阅前面第8章的“闭包作为函数模板”一节。

下面还有一组更怪异的类。从包名应该能得到一点线索，可以从中了解它们要做什么。

```

use strict;
package Tie::DevNull;

    sub TIEHANDLE {
        my $class = shift;
        my $fh = local *FH;
        bless \$fh, $class;
    }
    for (qw(READ READLINE GETC PRINT PRINTF WRITE)) {
        no strict "refs";
        *$_ = sub { return };
    }

package Tie::DevRandom;

    sub READLINE { rand() . "\n" }
    sub TIEHANDLE {
        my $class = shift;
        my $fh = local *FH;
        bless \$fh, $class;
    }

    sub FETCH { rand() }
    sub TIESCALAR {
        my $class = shift;
        bless \my $self, $class;
    }

package Tie::Tee;

    sub TIEHANDLE {
        my $class = shift;
        my @handles;
        for my $path (@_) {
            open(my $fh, ">$path") || die "can't write $path";
            push @handles, $fh;
        }
        bless \@handles, $class;
    }

    sub PRINT {
        my $self = shift;
        my $ok = 0;
        for my $fh (@$self) {

```



```

        $ok += print $fh @_;
    }
    return $ok == @$self;
}

```

`Tie::Tee`类模拟标准Unix `tee(1)`程序，它把一个输出流发送到多个不同的目的地。`Tie::DevNull`类模拟null设备，即Unix系统上的`/dev/null`。`Tie::DevRandom`类生成随机数作为句柄或标量（取决于你调用TIEHANDLE还是TIESCALAR！）。可以如下调用这些方法：

```

package main;

tie *SCATTER, "Tie::Tee", qw(tmp1 - tmp2 >tmp3 tmp4);
tie *RANDOM,   "Tie::DevRandom";
tie *NULL,    "Tie::DevNull";
tie my $randy, "Tie::DevRandom";

for my $i (1..10) {
    my $line = <RANDOM>;
    chomp $line;
    for my $fh (*NULL, *SCATTER) {
        print $fh "$i: $line $randy\n";
    }
}

```

这会在屏幕上生成类似这样的结果：

```

1: 0.124115571686165 0.20872819474074
2: 0.156618299751194 0.678171662366353
3: 0.799749050426126 0.300184963960792
4: 0.599474551447884 0.213935286029916
5: 0.700232143543861 0.800773751296671
6: 0.201203608274334 0.0654303290639575
7: 0.605381294683365 0.718162304090487
8: 0.452976481105495 0.574026269121667
9: 0.736819876983848 0.391737610662044
10: 0.518606540417331 0.381805078272308

```

这还不是全部！结果会写到屏幕上是因为以上`*SCATTER` tie中包含有`-`。不过，这行代码还指出要创建文件`tmp1`、`tmp2`和`tmp4`，另外还要追加到文件`tmp3`（我们在循环中还写入了`*NULL`文件句柄，尽管当然这不会显示任何有趣的结果，除非你对黑洞感兴趣）。

## 一个解除绑定小陷阱

如果想使用由`tie`或`tied`返回的对象，而且这个类定义了一个析构函数，就得当心这里的一个小陷阱。考虑下面这个例子（必须承认，这是专门设计的一个例子），这个类使用一个文件来记录为一个标量赋的所有值：

```

package Remember;

sub TIESCALAR {

```

```

    my $class = shift;
    my $filename = shift;
    open(my $handle, ">", $filename)
        || die "Cannot open $filename: $!\n";
    print $handle "The Start\n";
    bless {FH => $handle, VALUE => 0}, $class;
}

sub FETCH {
    my $self = shift;
    return $self->{VALUE};
}

sub STORE {
    my $self = shift;
    my $value = shift;
    my $handle = $self->{FH};
    print $handle "$value\n";
    $self->{VALUE} = $value;
}

sub DESTROY {
    my $self = shift;
    my $handle = $self->{FH};
    print $handle "The End\n";
    close $handle;
}

1;

```

下面是使用这个Remember类的一个例子：

```

use strict;
use Remember;

my $fred;
$x = tie $fred, "Remember", "camel.log";
$fred = 1;
$fred = 4;
$fred = 5;
untie $fred;
system "cat camel.log";

```

执行时会得到以下输出：

```

The Start
1
4
5
The End

```

到目前为止还不错。下面再为Remember类增加一个方法，允许文件中有注释，如下：

```

sub comment {
    my $self = shift;
    my $message = shift;

```

```

        print { $self->{FH} } $handle $message, "\n";
    }

```

以下还是前面的例子，不过修改为使用comment方法：

```

use strict;
use Remember;

my ($fred, $x);
$x = tie $fred, "Remember", "camel.log";
$fred = 1;
$fred = 4;
comment $x "changing...";
$fred = 5;
untie $fred;
system "cat camel.log";

```

现在文件将为空，这可能不是你想要的结果。下面来解释这是为什么。绑定一个变量会把它与构造函数返回的对象关联起来。这个对象通常只有一个引用：隐藏在tied变量后面。调用“untie”会打破这个关联，并消除这个引用。由于这个对象没有其他引用，所以会触发DESTROY方法。

不过，在上面的例子中，我们在\$x中存储了绑定对象的第二个引用。这说明，调用untie之后，这个对象还有一个合法的引用，因此不会触发DESTROY，文件也不会刷新输出和关闭。正是因为这个原因，结果没有输出：文件句柄的缓冲区仍在内存中。在程序退出之前并没有存储到磁盘上。

要检查这种情况，可以使用-w命令行标志，或者在当前词法作用域中包含use warnings "untie" pragma。这两个技术都能发现某个untie调用后绑定对象仍有其他引用。如果是这样，Perl会打印以下警告：

```

untie attempted while 1 inner references still exist

```

要让程序正常工作，而不显示警告，可以在调用untie之前消除绑定对象的所有其他引用。可以显式地消除这些引用：

```

undef $x;
untie $fred;

```

不过，通常只需要确保变量会在适当的时间出作用域就可以解决这个问题。

## CPAN上的模块

在你鼓足干劲想要写自己的绑定模块之前，应该先看看有没有人已经做了这个工作。CPAN上有很多绑定模块，而且每天还不断有新模块加入（嗯，也许是每个月）。表14-2列出了这样一些模块。



表14-2: CPAN上的绑定模块

模块	描述
<code>IO::WrapTie</code>	将绑定对象包装在 <code>IO::Handle</code> 接口中
<code>MLDBM</code>	在一个DBM文件中透明地存储复杂数据值，而不只是平面字符串
<code>Tie::Cache::LRU</code>	实现一个最近最少使用（LRU）缓存
<code>Tie::Const</code>	提供常量标量和散列
<code>Tie::Counter</code>	让标量变量在每次访问时增1
<code>Tie::CPHash</code>	实现一个保留大小写但大小写不敏感的散列
<code>Tie::Cycle</code>	通过一个标量循环处理一个值列表
<code>Tie::DBI</code>	将散列绑定到DBI关系数据库
<code>Tie::Dict</code>	将散列绑定到一个RPC字典服务器
<code>Tie::DictFile</code>	将散列绑定到一个本地字典文件
<code>Tie::DNS</code>	将接口绑定到 <code>Net::DNS</code>
<code>Tie::EncryptedHash</code>	包含加密字段的散列（和基于散列的对象）
<code>Tie::FileLRUCache</code>	实现一个轻量级、基于文件系统的持久LRU缓存
<code>Tie::FlipFlop</code>	实现一个在两个值之间切换的绑定
<code>Tie::HashDefaults</code>	允许散列有默认值
<code>Tie::HashHistory</code>	跟踪散列的所有变更历史
<code>Tie::iCal</code>	将iCal文件绑定到Perl散列
<code>Tie::IxHash</code>	为Perl提供有序关联数组
<code>Tie::LDAP</code>	为LDAP数据库提供一个接口
<code>Tie::Persistent</code>	通过 <code>tie</code> 实现持久数据结构
<code>Tie::Pick</code>	从一个集合随机选择（和删除）一个元素
<code>Tie::RDBM</code>	将散列绑定到关系数据库
<code>Tie::STDERR</code>	将STDERR的输出发送到另一个进程（如一个邮件程序）
<code>Tie::Syslog</code>	绑定一个文件句柄来自动记录其输出
<code>Tie::TextDir</code>	绑定一个文件目录
<code>Tie::Toggle</code>	<code>False</code> 和 <code>true</code> 交替，如此循环反复
<code>Tie::TZ</code>	绑定 <code>\$TZ</code> ，设置 <code>%ENV</code> 并调用 <code>tzset(3)</code>
<code>Tie::VecArray</code>	为一个位向量提供数组接口
<code>Tie::Watch</code>	在Perl变量上设置观察点
<code>Win32::TieRegistry</code>	提供强大而便利的方法来管理一个Microsoft Windows注册表

# Perl的技术





# 进程间通信

计算机进程之间的通信方式几乎与人与人之间的交流方式一样多。绝对不能低估进程间通信的难度。如果你的朋友在使用肢体语言，而你只注意语言线索是没有用的。类似的，只有当两个进程已经协商好如何通信，而且遵循在此基础上建立的约定，它们才能正常地通信。对于各种不同类型的通信，从词法到实用层次，从“要使用哪种语言”到“轮到谁来发言”，所有这些都需要达成约定。这些约定是必不可少的，因为如果没有具体的上下文，很难交流纯粹的语义。

按我们的说法，进程间通信通常拼作IPC（interprocess communication）。Perl提供了不同层次的IPC工具，有非常简单的，也有非常复杂的。使用哪个工具取决于要交流的信息的复杂程度。最简单的信息几乎就是根本没有信息：只是知道某个特定的事件在某个特定的时间点发生。在Perl中，这些事件通过一种信号机制（模拟Unix的信号系统）进行通信。

在另一个极端，Perl的套接字（socket）允许你使用你喜欢的任何协议（要求双方都支持）与互联网上的任何其他进程通信。很自然的，这种自由是有代价的：你必须完成很多步骤来建立连接，并确保你与另一端进程都讲同样的语言。这就要求你必须遵守很多其他奇怪的惯例（取决于本地约定）。要保证协议正确，甚至还要求你必须讲XML、Java或Perl之类的语言。真是很恐怖。

在这两个极端之间还有另外一些工具，主要用于与同一台机器上的进程通信。这包括老式文件、管道、FIFO和各种System V IPC系统调用。不同平台对这些工具的支持也有所不同。现代Unix系统（包括Apple的Mac OS X）应该支持所有这些工具，另外除了信号和SysV IPC外，其余的大多数工具（包括管道、子进程创建、文件锁定和套接字）在所有较新的Microsoft操作系统上都会得到支持<sup>注1</sup>。

注1： 嗯，除了AF\_UNIX套接字以外。

关于移植的更多一般性信息可以在标准Perl文档集中找到（你的系统会用它选择的某种格式来显示），具体参见*perlport*。与Microsoft有关的信息可以参阅*perlwin32*和*perlfork*，甚至在一些非Microsoft系统上也安装了这些文档。我们还推荐以下参考书：

- Tom Christiansen和Nathan Torkington所著的《Perl Cookbook》第2版（O'Reilly出版），第16章到第18章。
- W. Richard Stevens所著的《Advanced Programming in the UNIX Environment》（Addison-Wesley出版）。
- W. Richard Stevens所著的《TCP/IP Illustrated》第1卷到第3卷（Addison-Wesley出版）。

## 信号

Perl使用一种很简单的信号处理模型：`%SIG`散列包含用户自定义信号处理器的引用（包括符号引用或硬引用）。某些事件会导致操作系统向受该事件影响的进程发送一个信号。此时会调用响应这个事件的处理器，并提供一个参数，其中包含触发方法调用的信号的名字。要向另一个进程发送一个信号，可以使用`kill`函数。可以认为它在向另一个进程发送一个包含1个二进制位的信息<sup>注2</sup>。如果该进程已经为这个信号安装了一个信号处理器，接收到这个信号时就可以执行处理器代码。不过，发送进程没有办法得到任何返回值，它只知道这个信号已经正常发出。发送者不会接收任何反馈来告诉它接收进程（如果有）对这个信号做何处理。

我们把这个工具归类为一种IPC，不过，实际上信号可以来自不同的来源，而不只是其他进程。信号可以来自你自己的进程，用户在键盘上输入某个特定的序列时（如Control-C或Control-Z）也可能生成信号，另外发生某个特殊的事件时（如一个子进程退出，或者进程用尽了所有栈空间或已经达到某个文件大小或内存限制），内核也可能产生一个信号。不过，你自己的进程能很容易地区分这些情况。信号就像是一个送到你家门口的神秘包裹，而且包裹上没有写返回地址。打开时最好小心一点。

由于`%SIG`数组中的元素可能是硬引用，所以通常的做法是使用匿名函数作为简单信号处理器：

```
$SIG{INT} = sub { die "\nOutta here!\n" };
$SIG{ALRM} = sub { die "Your alarm clock went off" };
```

或者，你可以创建一个命名函数，把它的名字或引用赋至散列的适当的槽中。例如，要截

---

注2：实际上，往往是5个或6个二进制位，取决于你的操作系统定义了多少个信号，另外还要看其他进程是否利用了你未发送不同信号这一事实。



获中断和退出信号（通常绑定到键盘上的Control-C和Control-\序列），可以建立类似下面的处理器：

```
sub catch_zap {
    my $signame = shift();
    our $shucks++;
    die "Somebody sent me a SIG$signame!";
}
$shucks = 0;
$SIG{INT} = "catch_zap";      # 总表示&main::catch_zap
$SIG{INT} = \&catch_zap;      # 最佳策略
$SIG{QUIT} = \&catch_zap;     # 再捕获另一个信号
```

注意，在这个信号处理器中，我们只是设置了一个全局变量，然后用die产生一个异常。在Perl提供安全信号之前，这一点很重要，因为在大多数系统上C库都是不可再入的，而且信号会异步发送。因此，即使Perl代码表现很好，仍有可能导致内核转储。如果使用安全信号，就不存在这些问题了。

要捕获信号，一种更容易的方法是使用sigtrap pragma安装简单的默认信号处理器：

```
use sigtrap qw(die INT QUIT);
use sigtrap qw(die untrapped normal-signals
               stack-trace any error-signals);
```

如果你不想写你自己的处理器，但是又想捕获一些危险的信号，并且相应地完成关闭动作，那么这个pragma会很有用。默认情况下，有些信号对你的进程来说非常致命，程序接收到这样一个信号时只能停止。很遗憾，这意味着它不会调用END函数来完成退出时处理，也不会调用DESTROY方法来完成对象最终化处理。不过，对于普通的Perl异常（如调用die时抛出的异常），则会调用这些方法，所以可以使用这个pragma轻松地把信号转换为异常。尽管不是你自己处理信号，但你的程序仍会有正确的表现。参见第29章关于use sigtrap的描述，可以从中了解这个pragma的更多特性。

还可以把%SIG处理器设置为字符串“IGNORE”或“DEFAULT”，如果是这样，Perl会尝试丢弃这个信号，或者完成对应这个信号的默认动作（不过有些信号既不能捕获也不能忽略，如KILL和STOP信号，参见signal(3)，其中列出了你的系统提供了哪些可用的信号以及相应的默认行为）。

操作系统认为信号是一个数字而不是名字，不过与大多数人一样，Perl更喜欢符号名而不是魔法数。要找到信号的名字，可以列出%SIG散列的键，或者使用kill -l 命令（如果你的系统中有这个命令）。还可以使用Perl的标准Config模块来确定你的操作系统在信号名和信号数之间建立的映射。可以参考Config(3)，其中提供了这样一个例子。

由于%SIG是一个全局散列，为这个散列赋值会影响你的整个程序。对于程序的其余部分，更贴心的做法是把信号捕获限制在一个有限的范围内。要达到这个目的，可以使用一个局



部信号处理器赋值，一旦退出外围代码块，这个局部信号处理器就会失去作用（不过，要记住，在这个代码块调用的函数中local值仍是可见的）。

```
{
    local $SIG{INT} = "IGNORE";
    ...      # 在这里做你想做的任何事情，可以忽略所有SIGINT。
    fn();    # fn()中也会忽略SIGINT!
    ...      # 还有这里。
}           # 退出这个块会恢复原来的$SIG{INT}值。

fn();      # fn()中（可能）不再忽略SIGINT。
```

## 向进程组发送信号

进程（至少在Unix下）会组织为进程组，通常对应整个作业。例如，执行一个shell命令时，如果它由一系列过滤器命令组成，通过管道将数据从一个进程传送到另一个进程，那么这些进程（及其子进程）都属于同一个进程组。这个进程组有一个编号，对应于进程组领导进程（process group leader）的进程号。如果向一个正进程号发送信号，就是将信号发送到这个进程。不过，如果向一个负进程号发送信号，则是将该信号发送给每一个进程组号为相应正数（该负数的绝对值）的进程。也就是说，这些进程的进程组号为进程组领导进程的进程号（很方便地，进程组领导进程的进程组ID就是\$\$）。

假设你的程序要向由它直接启动的所有子进程（以及这些子进程启动的所有孙进程，还有这些孙进程启动的所有曾孙进程，依此类推）发送一个挂起信号。为此，程序首先调用setpgroup(0,0)，成为一个新的进程组的领导进程，它创建的所有进程就会属于这个新的进程组。这些进程可能是通过fork手动启动，也可能通过管道式open自动启动，或者由系统作为后台作业启动（"cmd &"），究竟以何种方式启动并不重要。即使这些进程有它们自己的子进程，向整个进程组发送一个挂起信号时能够找到所有这些进程（但有些进程除外，如果一些进程另外设置了自己的进程组，或者改变了已保存UID或有效UID，而与你的真实UID或有效UID不再匹配，这些进程就不会收到信号，这使它们对这些信号有一种“外交豁免权”）。

```
{
    local $SIG{HUP} = "IGNORE";      # 免除自己
    kill(HUP, -$);                   # 向我自己的进程组发信号
}
```

另一个有意思的信号是信号0。它实际上并不影响目标进程，只是检查该进程是否活动以及是否改变了UID。也就是说，它会检查是否可以向目标进程发送一个信号，但并不真正发送信号。

```
unless (kill 0 => $kid_pid) {
    warn "something wicked happened to $kid_pid";
}
```

Perl从Unix移植到Microsoft时，相对于Unix版本，信号0是唯一一个在两个平台上作用相同的信号。在Microsoft系统上，kill并不真正发送一个信号。实际上，它会强制目标进程退出，退出状态由信号数指示。可能将来某一天会修正这个问题。不过，这个神奇的信号0仍将表现出标准、非破坏性的特点。

## 俘获僵尸进程

退出一个进程时，会由内核向其父进程发送一个CHLD信号，这个进程会变成一个僵尸进程（zombie）<sup>注3</sup>，直到父进程调用wait或waitpid。如果在Perl中使用其他方式（而不是调用fork）启动另一个进程，Perl会负责俘获僵尸子进程；不过如果你使用fork来创建子进程，就得自己负责清理。在很多（但不是全部）内核上，自动俘获僵尸的一种简单方法是将\$SIG{CHLD}设置为"IGNORE"。一种更灵活（但也更烦琐）的方法是由你自己来俘获。因为在你打算处理僵尸子进程时，可能不止一个子进程已经终止，必须通过一个循环来收集这些僵尸进程，直到再没有更多僵尸进程为止：

```
use POSIX ":sys_wait_h";
sub REAPER { 1 until waitpid(-1, WNOHANG) == -1 }
```

根据根据需要运行这个代码，可以为它设置一个CHLD信号处理器：

```
$SIG{CHLD} = \&REAPER;
```

或者如果在一个循环中运行，只需要经常调用这个俘获程序。

## 让慢操作超时

信号的一种常见用法是为运行时间很长的操作施加时间限制。如果在一个Unix系统上（或者支持ALRM信号的任何其他POSIX兼容的系统），可以请求内核在将来某个时间为进程发送一个ALRM信号：

```
use Fcntl ":flock";
eval {
    local $SIG{ALRM} = sub { die "alarm clock restart" };
    alarm 10;                # 设置10秒后报警
    eval {
        flock(FH, LOCK_EX)   # 一个阻塞的排他锁
        || die "can't flock: $!";
    };
    alarm 0;                 # 取消报警
};
alarm 0;                    # 竞态条件保护
die if $@ && $@ !~ /alarm clock restart/; # 重新启动
```

---

注3： 没错，这确实是一个技术术语。



如果等待锁时发生警报，倘若只是捕获这个信号并返回，就会回到flock，因为Perl会尽可能自动重启系统调用。如果不想这么做，唯一的办法就是通过die产生一个异常，然后让eval捕获这个异常（这是可行的，因为异常最后会调用C库中的longjmp(3)函数，它将避免重启系统调用）。

这里使用了嵌套的异常捕获，因为如果你的平台上没有实现flock，调用flock就会产生一个异常，因此要确保清除警报。提供第二个alarm 0是因为有时信号会在运行flock之后但在到达第一个alarm 0之前出现。如果没有第二个alarm，可能会遭遇一个小的竞态条件，不过在竞态条件下大小并不重要，我们关心的只是要么有竞争，要么没有竞争。当然我们还是希望不出现竞态条件。

## 阻塞信号

有时你可能希望执行关键代码段时延迟接收信号。你不想盲目地忽略这个信号，但是当前正在做的事情实在太重要，不能中断。Perl的%SIG散列没有实现信号阻塞，不过POSIX模块通过其sigprocmask(2)系统调用接口实现了信号阻塞：

```
use POSIX qw(:signal_h);
$sigset = POSIX::SigSet->new;
$blockset = POSIX::SigSet->new(SIGINT, SIGQUIT, SIGCHLD);
sigprocmask(SIG_BLOCK, $blockset, $sigset)
    || die "Could not block INT,QUIT,CHLD signals: $!\n";
```

一旦这3个信号都被阻塞，你就可以做你想做的任何事情，而不用担心受到干扰。执行完你的关键代码段时，可以恢复原来的信号掩码，取消这些信号的阻塞：

```
sigprocmask(SIG_SETMASK, $sigset)
    || die "Could not restore INT,QUIT,CHLD signals: $!\n";
```

在阻塞期间，如果某个信号到来，则会立即发送。如果有两个或更多不同的信号到来，信号发送的顺序并不确定。另外，阻塞时某个特定信号接收到一次和多次并没有区别<sup>注4</sup>。例如，阻塞CHLD信号时如果9个子进程退出，解除阻塞之后仍然只会调用一次处理器（如果有的话）。正是因为这个原因，俘获僵尸进程时一定要循环处理，直到再没有更多僵尸进程为止。

## 信号安全性

在v5.8之前，Perl会把信号当作中断并立即处理，而不论解释器当前的状态。由于存在再入问题，这会有固有的不可靠性。有可能会破坏Perl自己的内存，更糟糕的是，你的进程可能会崩溃。

---

注4：通常是这样。一些实时系统可能根据最新规范实现了可计数信号，不过我们还没有看见过。



如今，有信号到达你的进程时，Perl只是做一个标志，指示信号到来。然后再在解释器循环的下一个安全点处理所有到来的信号。这是安全的、有序的，而且是可靠的，不过不一定及时。Perl的一些操作码执行的时间可能很长，如在一个极大的列表上调用`sort`。

要让Perl仍然采用原来不可靠的方式处理（或错误地处理）信号，可以把`PERL_SIGNALS`环境变量设置为“unsafe”。不过，最好先好好读一读`perlipc`手册页中“延迟信号”一节。

## 文件

以前你可能从未想过把文件当作一种IPC机制，不过文件在进程间通信中确实占有很大的份额，甚至比所有其他方式加在一起的份额还要多。如果一个进程将它的珍贵数据存入一个文件，另一个进程随后获取这个数据，这两个进程就进行了通信。这里介绍的所有IPC形式中，文件提供了其他方式所没有的一些特有特点：就像在沙漠里深藏了千年的手卷，可以把文件储藏起来，在其作者寿终正寝很久以后再来读取<sup>注5</sup>。考虑到这种持久性，再加上简单易用，文件一直如此流行也就不足为奇了。

使用文件从已经消亡的过去向未知的将来传递信息并没有太多让人惊奇的地方。可以把文件写到某种持久存储的介质上，如磁盘，仅此而已（如果文件中包含HTML，可能要告诉Web服务器到哪里查找这个文件）。真正有意思的是通信各方都是“活动的”，而且希望相互通信，这才是难点。如果没有达成协议来明确轮到谁发言，根本不可能建立可靠的通信。这种协议可以通过文件锁定来得到，下一节会介绍有关内容。在此之后，我们还会讨论父进程与其子进程之间存在的特殊关系，利用这种特殊关系，相关各方可以通过对相同文件的继承访问来实现信息交换。

在远程访问、同步、可靠性和会话管理等方面，文件当然存在其局限性。这一章的其他小节还会介绍能够克服这些局限性的其他IPC机制。

## 文件锁定

在一个多任务环境中，如果你在使用某个文件，需要特别当心不要与试图使用同一个文件的其他进程冲突。只要所有进程都只是读文件，那就没有问题。不过，即使只有一个进程需要写文件，除非提供某种锁定机制充当交通警察的角色，否则很可能出现混乱。

绝对不要把文件名是否存在（即`-e $file`）当作指示锁定的手段，因为测试该文件名是否存在与对这个文件的操作（如创建、打开或解除链接）之间会存在一种竞态条件。有关的更多内容参见第20章“处理竞态条件”一节。

Perl的可移植锁定接口是`flock(HANDLE, FLAGS)`函数，参见第27章的描述。Perl只使用了绝

---

注5： 假设进程可以寿终正寝。

大多数平台上广泛都有的最简单的锁定特性，从而提供最大的可移植性。这些语义很简单，大多数系统都可以模拟，也包括那些不支持传统同名系统调用的系统，如System V或Windows NT（不过，如果你运行的Microsoft系统比NT还要早，可能就不那么走运了，Mac OS X以前的苹果系统可能也不提供支持）。

锁有两类：共享锁（LOCK\_SH标志）和排他锁（LOCK\_EX标志）。尽管“排他”（exclusive）听上去有“绝对”的意思，不过并不要求进程必须服从文件的锁。也就是说，flock只实现建议性锁定（advisory locking），这表示锁定一个文件并不会阻止其他进程读甚至写这个文件。对于进程来说，请求一个排他锁只是让操作系统将它挂起，直到当前的所有锁（不论是共享锁还是排他锁）持有者完成操作。类似地，进程请求一个共享锁时，它也只是将自己挂起，直到再没有其他排他锁持有者为止。只有当各方都使用文件锁定机制时才能安全地访问一个大家都在竞争的文件。

因此，默认地flock是一个阻塞操作。也就是说，如果不能立即得到你想要的锁，操作系统会把你的进程挂起，直到你能得到锁为止。可以如下得到一个阻塞的共享锁，这通常用于读文件：

```
use Fcntl qw(:DEFAULT :flock);
open(FH, "< filename")      || die "can't open filename: $!";
flock(FH, LOCK_SH)          || die "can't lock filename: $!";
# 现在读取FH
```

可以在flock请求中包含LOCK\_NB标志，用一种非阻塞方式请求锁。如果不能立即得到锁，这个函数会失败，并立即返回false。下面给出一个例子：

```
flock(FH, LOCK_SH | LOCK_NB)
    || die "can't lock filename: $!";
```

这里我们只是抛出一个异常，除此以外，你可能还想做些其他处理，不过绝对不能对文件作任何I/O操作。如果你被拒绝，没有得到锁，就不能访问这个文件，直到你真正得到锁之后才能访问。在此之前，谁知道文件会处于一种怎样的混乱状态！非阻塞模式的主要目的是允许你在等待时做些其他工作。另外，这对于建立更友好的交互也很有用，可以告诉用户可能需要一些时间才能得到锁（以免用户觉得没人理睬他）：

```
use Fcntl qw(:DEFAULT :flock);
open(FH, "< filename") || die "can't open filename: $!";
unless (flock(FH, LOCK_SH | LOCK_NB)) {
    local $| = 1;
    print "Waiting for lock on filename...";
    flock(FH, LOCK_SH) || die "can't lock filename: $!";
    print "got it.\n"
}
# 现在读取FH
```

有些人可能想把这个非阻塞锁放在一个循环里。非阻塞模式的主要问题在于，再回过去检



查时，由于之前你放弃了你在队伍中的排位，锁可能已经被别人拿走了。有时你必须一直排队等待。幸运的话，还可以找本杂志看看。

锁加在文件句柄上，而不是加在文件名上<sup>注6</sup>。关闭文件时，不论是通过调用`close`显式关闭文件，还是重新打开句柄或者退出进程来隐式关闭文件，锁都会自动解除。

要得到一个排他锁（通常用于写文件），则要更加小心。不能用常规的`open`，如果使用打开模式`<`，倘若文件还不存在，打开文件就会失败，而如果使用打开模式`>`，倘若文件已经存在则会被清除。应当使用`sysopen`，这样在覆盖文件之前可以先加锁。一旦安全地打开文件来完成写入（不过还没有真正写内容），需要先成功地获取排他锁，然后才能删除文件数据。现在可以用新数据覆盖这个文件。

```
use Fcntl qw(:DEFAULT :flock);
sysopen(FH, "filename", O_WRONLY | O_CREAT)
    || die "can't open filename: $!";
flock(FH, LOCK_EX)
    || die "can't lock filename: $!";
truncate(FH, 0)
    || die "can't truncate filename: $!";
# 现在写FH
```

如果想“原地”修改一个文件的内容，也需要使用`sysopen`。这一次要同时请求读写访问，如果需要还可能要创建文件。文件打开后，不过在具体完成读写之前，需要得到排他锁，并在整个事务期间保留这个锁。通常最好通过关闭文件来释放锁，因为这样可以保证所有缓冲区在锁释放之前写入。

更新时需要读出旧值，并写入新值。必须在同一个排他锁下完成这两个操作，以免另一个进程在你读旧值之后（或者甚至之前）但在写新值之前读取这个值（很可能不正确）。这一章后面介绍共享内存时还会遇到这种情况。

```
use Fcntl qw(:DEFAULT :flock);

sysopen(FH, "counterfile", O_RDWR | O_CREAT)
    || die "can't open counterfile: $!";
flock(FH, LOCK_EX)
    || die "can't write-lock counterfile: $!";
$counter = <FH> || 0; # 第一次是undef
seek(FH, 0, 0)
    || die "can't rewind counterfile: $!";
```

---

注6：实际上，锁并不是加在文件句柄上，而是加在与文件句柄关联的文件描述符上，因为操作系统并不知道文件句柄。这说明，如果得到`die`消息指示未能得到某些文件名上的锁，这些说法从技术上讲都是不正确的。不过如果得到类似下面的错误消息，如“无法在一个文件描述符表示的文件上得到一个锁，这个文件描述符与原先按路径`filename`打开的文件句柄关联，尽管现在`filename`与句柄表示的可能是完全不同的文件”，这会让用户很困惑（更不用说读者了）。



```

print FH $counter+1, "\n"
    || die "can't write counterfile: $!";

# 在这个程序中，下面这行代码从技术上讲是多余的，
# 不过通常情况下这是一种很好的做法。
truncate(FH, tell(FH))
    || die "can't truncate counterfile: $!";
close(FH)
    || die "can't close counterfile: $!";

```

不能锁定一个尚未打开的文件，另外也不能对多个文件应用同一个锁。不过，可以使用一个完全独立的文件作为某种信号量，类似于交通灯，通过信号量文件上的常规共享锁和排他锁为某个文件提供受控访问。这种方法有很多优点。可以利用一个锁文件控制对多个文件的访问，从而避免死锁（如果一个进程试图以某种顺序锁定这些文件，而另一个进程想要以一种不同的顺序锁定这些文件，就会出现死锁）。另外可以使用一个信号量文件锁定整个文件目录。甚至在访问不在文件系统中的某个对象时，如一个共享内存对象或套接字（可能有多个预建的服务器想在这个套接字上调用accept），也可以利用信号量文件控制其访问。

如果有一个DBM文件，而且它没有提供自己的显式锁定机制，使用一个辅助的锁文件就是控制多代理并发访问的最佳方法。否则，DBM库的内部缓存可能与磁盘上的文件不同步。调用dbmopen或tie之后，需要打开并锁定信号量文件。如果使用O\_RDONLY打开数据库，可能要用LOCK\_SH加锁。否则可以使用LOCK\_EX实现更新数据库的排他访问（同样的，只有当所有参与者都同意注意这个信号量时这才有效）。

```

use Fcntl qw(:DEFAULT :flock);
use DB_File;          # 只用于演示，任何db都可以。

$DBNAME = "/path/to/database";
$LCK = $DBNAME . ".lockfile";

# 如果想把数据放在锁文件中，要使用O_RDWR
sysopen(DBLOCK, $LCK, O_RDONLY | O_CREAT)
    || die "can't open $LCK: $!";

# 打开数据库之前必须锁定
flock(DBLOCK, LOCK_SH)
    || die "can't LOCK_SH $LCK: $!";

tie(%hash, "DB_File", $DBNAME, O_RDWR | O_CREAT)
    || die "can't tie $DBNAME: $!";

```

现在可以安全地对绑定%hash做你想做的任何事情。处理完数据库后，要确保显式地释放这些资源，而且要以与获取资源相反的顺序逐一释放：

```

untie %hash;          # 必须在关闭锁文件之前关闭数据库
close DBLOCK;         # 现在可以安全地释放锁了

```

如果安装了GNU DBM库，可以使用标准GDBM\_File模块的隐式锁定。除非开始的tie包含GDBM\_NOLOCK标志，否则这个库会确保一次只有一个写者可以打开GDBM文件，而且读者和写者不能同时让数据库处于打开状态。

## 传递文件句柄

只要使用fork创建一个子进程，这个新进程就会继承其父进程的所有打开的文件句柄。要展示使用文件句柄完成进程间通信，最容易的做法是首先使用平面文件。要了解这是如何做到的，管道和套接字（本章后面将会介绍）的机制更复杂，而了解平面文件是掌握那些更复杂机制的基础。

这个最简单的例子首先打开一个文件，并建立一个子进程。然后这个子进程使用已经为它打开的文件句柄：

```
open(INPUT, "< /etc/motd") || die "/etc/motd: $!";
if ($pid = fork) { waitpid($pid,0) }
else {
    defined($pid) || die "fork: $!";
    while (<INPUT) { print "$.: $_" }
    exit; # 不要让子进程又落回到主代码
}
# 在父进程中INPUT句柄现在位于文件末尾 (EOF)
```

一旦open授权允许访问一个文件，这个授权会一直保留，直到文件句柄关闭；如果改变文件权限，或者改变所有者的访问权限，这对于文件的访问性没有任何影响。即使以后进程修改了它的用户ID或组ID，或者文件将其所有权改为属于另一个用户或组，这些也不会影响已经打开的文件句柄。提升权限后运行的程序（如set-id程序或系统守护进程）通常会在其提升权限下打开文件，然后把这个文件句柄交给一个子进程，否则这个子进程可能无法自己打开这个文件。

如果有意识地使用这个特性，它会带来很大方便，不过如果文件句柄意外地从一个程序漏到另一个程序，则会带来安全性问题。为了避免允许隐式地访问所有可能的文件句柄，一旦使用exec显式地执行一个新程序，或者通过管道式open、system或qx//（反引号）调用隐式执行一个程序，Perl都会自动关闭它打开的所有文件句柄（包括管道和套接字）。系统文件句柄STDIN、STDOUT和STDERR除外，因为它们的主要目的是在程序之间提供通信链路。所以要向一个新程序传递文件句柄，一种方法就是把这个文件句柄复制到以上某个标准文件句柄：

```
open(INPUT, "< /etc/motd")    || die "/etc/motd: $!";
if ($pid = fork) { wait }
else {
    defined($pid)              || die "fork: $!";
    open(STDIN, "<&INPUT")      || die "dup: $!";
    exec("cat", "-n")          || die "exec cat: $!";
}
```

如果确实希望新程序访问某个文件句柄，而不是以上的3个标准文件句柄，这也是可以的，不过必须完成下面两件事之一。Perl打开一个新文件（或管道或套接字）时，它会检查`$^F`（`$SYSTEM_FD_MAX`）变量的当前设置。如果这个新文件句柄使用的数字文件描述符大于`$^F`，则标志这个描述符为将要关闭的文件。否则，Perl保留其原有设置，所执行的新程序将继承对文件的访问。

预测新打开的文件句柄的文件描述符通常并不容易，不过可以在`open`期间将系统最大文件描述符临时设置为某个非常大的数：

```
# 打开文件，并标志INPUT在exec期间保持为打开状态
{
    local $^F = 10_000;
    open(INPUT, "< /etc/motd") || die "/etc/motd: $!";
} # 作用域退出时恢复$^F原来的值
```

现在你要做的就是让这个新程序注意刚打开的文件句柄的描述符编号。（在提供支持的系统上）最简洁的方案是传递一个相当于文件描述符的特殊文件名。如果你的系统有一个名为`/dev/fd`或`/proc/$$/fd`的目录，其中包含的文件分别从0编号到系统支持的最大描述符数，就可以使用这个策略（很多Linux操作系统同时包含这两个目录，不过只有`/proc`目录可以正确填充。BSD和Solaris更倾向于使用`/dev/fd`。你最好仔细检查你的系统，看看哪一个更适合你）。首先，利用前面的代码打开文件句柄，将它标志为在`exec`期间一直保持打开状态，然后如下创建进程：

```
if ($pid = fork) { wait }
else {
    defined($pid) || die "fork: $!";
    $fdfile = "/dev/fd/" . fileno(INPUT);
    exec("cat", "-n", $fdfile) || die "exec cat: $!";
}
```

通过使用`fcntl`系统调用，可以手动地设置文件句柄的`close-on-exec`（执行时关闭）标志。有时你创建文件句柄时可能没有意识到希望与子进程分享，这种情况下这种方法就会很方便。

```
use Fcntl qw/F_SETFD/;

fcntl(INPUT, F_SETFD, 0)
|| die "Can't clear close-on-exec flag on INPUT: $!\n";
```

还可以强制关闭一个文件句柄：

```
fcntl(INPUT, F_SETFD, 1)
|| die "Can't set close-on-exec flag on INPUT: $!\n";
```

也可以查询当前状态：

```
use Fcntl qw/F_SETFD F_GETFD/;
```



```
printf("INPUT will be %s across execs\n",
      fcntl(INPUT, F_GETFD, 1) ? "closed" : "left open");
```

如果你的系统不支持文件系统中指定的文件描述符，而且你想传递STDIN、STDOUT或STDERR以外的一个文件句柄，这仍然可以做到，不过必须对程序做些特殊的安排。对此，通常的做法是通过一个环境变量或命令行选项传递这个描述符编号。

如果所执行的程序是用Perl编写的，可以使用open将文件描述符转换为文件句柄。并不是指定一个文件名，而是使用“&=”后面跟一个描述符编号。

```
if (($ENV{input_fdno} // "") =~ /\^d$/) {
    open(INPUT, "<&=$ENV{input_fdno}")
    || die "can't fdopen $ENV{input_fdno} for input: $!";
}
```

如果你要运行一个Perl子例程，或者要运行的程序需要一个文件名参数，那就更容易了。Perl的常规open函数（而不是sysopen或三参数的open）提供了一个描述符打开（descriptor-opening）特性，可以利用这个特性自动完成。假设有以下这个简单的Perl程序：

```
#!/usr/bin/perl -p
# nl - 输入行编号
printf "%6d ", $.;
```

假设你已经设置INPUT句柄在exec期间保持打开状态，可以如下调用这个程序：

```
$fdspec = "<&=" . fileno(INPUT);
system("nl", $fdspec);
```

或者如下捕获输出：

```
@lines = `nl '$fdspec'`; # 单引号保护spec不被shell代换
```

不论是否用exec执行另一个程序，倘若使用从fork继承的文件描述符，会有一个小小的收获。与通过fork复制的变量不同（尽管复制得到的变量完全相同，但它们是独立的副本），两个进程中的文件描述符实际上是同一个描述符。一个进程从句柄读数据时，另一个进程中的定位指针（文件位置）也会前进，该数据对两个进程都不再可用。如果它们轮流读数据，会在文件中交替地读。这对于与串行设备、管道或套接字关联的句柄来说很直观，因为它们往往是提供短期数据的只读设备。不过，这种行为对于磁盘文件就不适用了。如果这里有问题，使用fork创建进程之后可以重新打开需要单独跟踪的文件。

fork操作符是一个源自Unix的概念，这说明并非在所有非Unix/非POSIX平台上都能正确地实现这个操作。特别需要说明，只有在Windows 98（或之后版本）上运行Perl v5.6（或更高版本）时，fork才能在Microsoft系统上正常工作。尽管在这些系统上可以通过同一个程序中的多个并发执行流来实现fork，不过这些不是那种可以默认共享所有数据的线程。这里只共享文件描述符。

# 管道

管道 (pipe) 是一个单向I/O通道, 可以把一个字节流从一个进程传输到另一个进程。管道包括命名管道和匿名管道两种。你可能对匿名管道比较熟悉, 所以我们先介绍这个内容。

## 匿名管道

向open的第二个参数末尾或开头追加一个管道符号时, Perl的open函数会打开一个管道而不是文件, 相应地会把其余的参数转换为一个命令, 这会解释为一个进程 (或进程集), 你希望通过管道为这个进程传入或传出一个数据流。可以如下建立一个写数据的子进程:

```
open SPOOLER, "| cat -v | lpr -h 2>/dev/null"
|| die "can't fork: $!";
local $SIG{PIPE} = sub { die "spooler pipe broke" };
print SPOOLER "stuff\n";
close SPOOLER || die "bad spool: $! $?";
```

这个例子实际上建立了两个进程, 我们会直接写入其中第一个 (运行`cat`) 进程。第二个进程 (运行`lpr`) 再接收第一个进程的输出。在shell编程中, 这通常称为一个管线 (pipeline)。管线可以有连续的多个进程, 前提是中间的进程知道如何表现得像过滤器 (filters), 也就是说, 它们会读取标准输入, 并写标准输出。

如果管道命令包含shell关心的特殊字符, Perl会使用默认的系统shell (Unix上是`/bin/sh`)。如果你只是启动一个命令, 并不需要 (或者不想) 使用shell, 那么可以使用一个多参数形式的管道式open调用:

```
open SPOOLER, "|-", "lpr", "-h" # requires 5.6.1
|| die "can't run lpr: $!";
```

如果将程序的标准输出作为另一个程序的管道重新打开, 以后打印到STDOUT的所有输出都将成为这个新程序的标准输入。所以如果要对程序的输出分页<sup>注7</sup>, 可以使用以下代码:

```
if (-t STDOUT) { # 仅当stdout是终端时
    my $pager = $ENV{PAGER} || "more";
    open(STDOUT, "| $pager") || die "can't fork a pager: $!";
}
END {
    close(STDOUT) || die "can't close STDOUT: $!"
}
```

写入与一个管道连接的文件句柄时, 完成处理后一定要用close显式地关闭这个句柄。这样主程序才不会在其子进程之前退出。

可以如下建立一个读数据的子进程:

---

注7: 也就是说, 一次显示一屏内容, 而不是一古脑完全输出。



```

open STATUS, "netstat -an 2>/dev/null |"
    || die "can't fork: $!";
while (<STATUS>) {
    next if /^(tcp|udp)/;
    print;
}
close STATUS || die "bad netstat: $! $?";

```

类似于输出，可以打开一个多级管线来完成输入。同样地，与前面一样，可以使用另外一种形式的`open`避免执行shell：

```

open STATUS, "-|", "netstat", "-an" # requires 5.6.1
    || die "can't run netstat: $!";

```

不过，这样就无法得到I/O重定向、通配符扩展或多级管道，因为Perl要依赖shell来完成这些工作。

你可能已经注意到，可以使用反引号来实现与打开管道读数据同样的效果：

```

print grep { !/^(tcp|udp)/ } `netstat -an 2>&1`;
die "bad netstat" if $?;

```

尽管反引号非常方便，不过它们必须一次把全部内容都读入内存，所以通常更高效的方法是打开你自己的管道文件句柄，一次一行或一个记录地读取这个文件。这就允许你更细粒度地控制整个操作，如果愿意还可以提前结束子进程。另外还可以一边接收一边处理输入，这样会更高效，因为两个或多个进程同时运行时计算机可以交替地完成多个操作（即使是在一个单CPU机器上，CPU正在做其他工作的同时，也可以完成输入和输出操作）。

由于你在并发地运行两个或多个进程，在`open`和`close`之间的任意时刻子进程都有可能面临灭顶之灾。这说明父进程必须检查`open`和`close`的返回值。只检查`open`还不够，因为这只会告诉你进程创建是否成功，以及后续的命令是否成功发出（只是最近的Perl版本才提供这个特性，而且只有当命令由`fork`创建的子进程直接执行而不是通过shell执行时才能做到这一点）。在此之后发生的所有灾难都会作为一个非0退出状态从子进程报告给父进程。`close`函数看到这个报告时，就会知道要返回一个`false`值，指示实际的状态值应当从 `$?` （`$CHILD_ERROR`）变量读取。所以检查`close`的返回值与检查`open`同样重要。如果写入一个管道，还要做好准备处理PIPE信号，如果你还没有完成数据发送，而另一端的进程已经结束，就会向你发送这个信号。

## 自言自语

IPC的另一种方法是让你的程序与自己对话，也就是一种自言自语。实际上，你的进程可以通过管道与自己的一个派生副本通信。这与上一节介绍过的管道式`open`很类似，只不过子进程会继续执行你的脚本而不是另外某个命令。

要让`open`函数这样做，需要使用一个包含负号的伪命令。所以`open`的第二个参数将类似于



“|-” 或 “|”，这取决于你希望通过管道从你自己发出数据还是把数据发送给你自己。与正常的fork命令一样，在父进程中open函数会返回子进程的进程ID，而在子进程中会返回0。另外还有一个不对称性，open指定的文件句柄只在父进程中使用。管道中子进程一端会根据实际情况关联到STDIN或STDOUT。也就是说，如果用|-打开一个“到负号”管道，可以向打开的文件句柄写数据，而子进程会在STDIN中看到你写入的数据：

```
if (open(TO, "|-")) {
    print TO $fromparent;
}
else {
    $tochild = <STDIN>;
    exit;
}
```

如果用-|打开一个“从负号”管道，可以从你打开的文件句柄读数据，这会返回子进程写入STDOUT的数据：

```
if (open(FROM, "-|")) {
    $toparent = <FROM>;
}
else {
    print STDOUT $fromchild;
    exit;
}
```

这种方法有一个常见的应用：希望从一个命令打开管道时可以利用这个方法绕过shell。之所以想这么做，原因可能是你传递给命令的文件名中可能会出现元字符，而你希望shell解释这些元字符。如果运行v5.6.1或更新版本，可以使用多参数形式的open来得到同样的结果。

这种创建子进程的open还可以用来安全地打开一个文件或命令，即使在一个假想UID或GID下运行。你创建的子进程会放弃所有特殊的访问权限，然后安全地打开文件或命令，并作为一个中间节点在更强大的父进程和它打开的文件或命令之间传递数据。可以在第20章“降权限访问命令和文件”一节看到有关的一些例子。

这种open的一种创造性用法是可以过滤你自己的输出。有些算法用两趟实现要比只用一趟实现容易得多。下面给出一个简单的例子，在这个例子中，我们模拟了Unix *tee*(1)程序，沿管道发送正常的输出。管道另一端的代理（我们自己的一个子例程）将输出分发给指定的所有文件：

```
tee("/tmp/foo", "/tmp/bar", "/tmp/glarch");

while (<>) {
    print "$ARGV at line $. => $_";
}
close(STDOUT) || die "can't close STDOUT: $!";
```

```

sub tee {
    my @output = @_;
    my @handles = ();
    for my $path (@output) {
        my $fh; # open会将其填入
        unless (open ($fh, ">", $path)) {
            warn "cannot write to $path: $!";
            next;
        }
        push @handles, $fh;
    }

    # 在父进程中重新打开STDOUT并返回
    return if my $pid = open(STDOUT, "|-");
    die "cannot fork: $!" unless defined $pid;

    # 在子进程中处理STDIN
    while (<STDIN>) {
        for my $fh (@handles) {
            print $fh $_ || die "tee output failed: $!";
        }
    }
    for my $fh (@handles) {
        close($fh) || die "tee closing failed: $!";
    }
    exit; # 不要让子进程返回到主代码!
}

```

这个技术可以重复应用，可以根据你的需要在输出流上放置多个过滤器。只需要调用函数以这种fork-open方式（即创建子进程的方式）打开STDOUT，让子进程从其父进程（被它看作是STDIN）读数据，并把处理过的输出继续传至流中的下一个函数。

用这种fork-open方式“自言自语”还有一个有意思的应用，如果一个糟糕的函数总是把结果不带任何格式地发送到STDOUT，利用这个方法可以捕获这个函数的输出。假设Perl只有printf而没有sprintf，你需要的可能是类似反引号的东西，不过要用Perl函数而不是外部命令实现：

```

badfunc("arg"); # 准备!
$string = forksub(\&badfunc, "arg"); # 作为字符串捕获
@lines = forksub(\&badfunc, "arg"); # 作为单独的行
sub forksub {
    my $kidpid = open my $self, "-|";
    defined $kidpid || die "cannot fork: $!";
    shift->(@_), exit unless $kidpid;
    local $/ unless wantarray;
    return <$self>; # 作用域退出时关闭
}

```

我们并没有说这样很高效，绑定文件句柄可能更快一些。不过，如果你比计算机工作还要繁忙，用这种方法写代码会容易得多。

## 双向通信

使用`open`通过管道连接到另一个命令对于实现单向通信很合适，那么双向通信呢？这种方法看起来很显然，但实际上行不通：

```
open(PROG_TO_READ_AND_WRITE, "| some program |") # 不正确!
```

如果你忘记启用警告，就会完全漏掉诊断消息：

```
Can't do bidirectional pipe at myprog line 3.
```

`open`函数不允许这样，因为除非特别当心，否则很容易出现死锁。不过，如果你决定确实要采用这种方法，可以使用标准`IPC::Open2`库模块将两个通道关联到一个子进程的STDIN和STDOUT。对于三向I/O还有一个`IPC::Open3`模块（它还允许捕获子进程的STDERR），不过这需要一个很麻烦的`select`循环，或者需要使用一个更方便的`IO::Select`模块。但是这样一来，你必须放弃Perl的缓冲输入操作，如`<> (readline)`。

下面是使用`open2`的一个例子：

```
use IPC::Open2;
local (*Reader, *Writer);
$pid = open2(\*Reader, \*Writer, "bc -l");
$sum = 2;
for (1 .. 5) {
    print Writer "$sum * $sum\n";
    chomp($sum = <Reader>);
}
close Writer;
close Reader;
waitpid($pid, 0);
print "sum is $sum\n";
```

还可以自动生成词法作用域文件句柄：

```
my ($fhread, $fhwrite);
$pid = open2($fhread, $fhwrite, "cat -u -n");
```

这种方法的普遍问题是实际上标准I/O缓冲会误事。尽管你的输出文件句柄会自动刷新（库会为你做到），使得另一端的进程能及时地得到你的数据，但是你无法让它也以同样的方式返回数据。我们的情况很特殊，非常幸运的是：`bc`也希望通过管道处理，并且知道要刷新输出每一个输出行。不过很少有命令是这样设计的，所以除非你自己编写双端管道另一端的程序，否则这种方法往往并不奏效。甚至简单的交互式程序（如`ftp`）在这里也不能正确工作，因为它们不会在管道上完成行缓冲，而只在tty设备上这么做。

在这方面，CPAN上的`IO::Pty`和`Expect`模块会有帮助，因为它们提供了一个真正的tty设备（实际上，是一个真正的伪tty，不过它就相当于一个真正的tty设备）。这会在另一个进程中完成行缓冲而不用修改程序。



如果把程序分为多个进程，希望它们都能进行双向通信，则不能使用Perl的高层管道接口，因为它们都是单向的。需要使用两个底层pipe函数调用，分别处理一个方向的会话：

```
pipe(FROM_PARENT, TO_CHILD)    || die "pipe: $!";
pipe(FROM_CHILD, TO_PARENT)    || die "pipe: $!";
select(((select(TO_CHILD), $| = 1))[0]);      # 自动刷新
select(((select(TO_PARENT), $| = 1))[0]);      # 自动刷新

if ($pid = fork) {
    close FROM_PARENT; close TO_PARENT;
    print TO_CHILD "Parent Pid $$ is sending this\n";
    chomp($line = <FROM_CHILD>);
    print "Parent Pid $$ just read this: '$line'\n";
    close FROM_CHILD; close TO_CHILD;
    waitpid($pid, 0);
} else {
    die "cannot fork: $!" unless defined $pid;
    close FROM_CHILD; close TO_CHILD;
    chomp($line = <FROM_PARENT>);
    print "Child Pid $$ just read this: '$line'\n";
    print TO_PARENT "Child Pid $$ is sending this\n";
    close FROM_PARENT; close TO_PARENT;
    exit;
}
```

在很多Unix系统上，实际上不必建立两个单独的pipe调用来实现父进程与子进程之间的全双工通信。socketpair系统调用可以在同一台机器上的相关进程之间提供双向连接。所以不必使用两个pipe，只需要一个socketpair就可以了。

```
use Socket;
socketpair(Child, Parent, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    || die "socketpair: $!";

# 或者让perl为你选择文件句柄
my ($kidfh, $dadfh);
socketpair($kidfh, $dadfh, AF_UNIX, SOCK_STREAM, PF_UNSPEC)
    || die "socketpair: $!";
```

fork创建子进程之后，父进程关闭Parent句柄，然后通过Child句柄读写。与此同时，子进程关闭Child句柄，再通过Parent句柄完成读写。

如果你想建立双向通信，因为要与你对话的进程实现了一个标准Internet服务，那么你应该跳过中间人，直接使用专门为此目的设计的CPAN模块（参见本章后面“套接字”一节，其中列出了这样一些模块）。

## 命名管道

命名管道（通常称为FIFO）是同一台机器上的无关进程之间建立会话的一种机制。“命名”管道中的名字存在于文件系统中，这只是一种有趣的说法，表示你可以把一个特殊的

文件放在某个文件系统命名空间中，这个文件系统命名空间底层是另外一个进程而不是一个磁盘<sup>注8</sup>。如果你希望一个进程连接到另外一个无关的进程，FIFO会很方便。打开一个FIFO时，你的进程会阻塞，直到另一端有进程为止。所以，如果一个读者首先打开一个FIFO，它会一直阻塞，直到写者出现。反之亦然。

要创建一个命名管道，可以使用POSIX `mkfifo`函数，如果你在运行一个POSIX系统，可以直接使用这个函数。在Microsoft系统上，可能要使用Win32::Pipe模块，它能创建命名管道，尽管表面看来与其作用正好相反（与我们其他人一样，Win32用户也使用`pipe`创建匿名管道）。

例如，假设你希望每次读你的`.signature`文件时生成一个不同的答案。只需要建立一个命名管道，另一端运行一个Perl程序来生成随机的结果。现在每次任何程序（如邮件程序、新闻阅读器、`finger`程序等）试图读取这个文件时，这个程序就会连接到你的程序，并读取一个动态的签名。

在下面的例子中，我们使用了很少见的`-p`文件测试操作符来确定是否有人（或程序）不小心删除了我们的FIFO<sup>注9</sup>。如果是这样，就没有必要再尝试打开文件，所以我们把它当作一个退出请求。如果使用了一个简单的`open`函数，模式设置为“> \$fpath”，则会出现一个很小的竞态条件，可能会意外地将签名创建一个平面文件（如果文件在`-p`测试和`open`之间消失不见）。也不能使用“+< \$fpath”模式，因为打开一个FIFO来完成读写是一种非阻塞打开方式（这一点只对FIFO适用）。可以使用`sysopen`并忽略`O_CREAT`标志来避免这个问题，而不会意外地创建一个文件。

```
use Fcntl;                # 要使用sysopen
chdir;                    # 回到home目录
$fpath = ".signature";
$ENV{PATH} .= ":usr/games";

unless (-p $fpath) {      # 不是一个管道
    if (-e _) {           # 而是另外某种方式
        die "$0: won't overwrite .signature\n";
    } else {
        require POSIX;
        POSIX::mkfifo($fpath, 0666) || die "can't mknod $fpath: $!";
        warn "$0: created $fpath as a named pipe\n";
    }
}

while (1) {
    # 如果手动删除了签名文件则退出
```

---

注8：对Unix域套接字可以做同样的处理，不过不能对它们使用`open`。

注9：另一种用法是查看一个文件句柄是否连接到一个管道（不论是命名管道还是匿名管道），如`-p STDIN`。



```

    die "Pipe file disappeared" unless -p $fpath;
    # 下一行阻塞，直到有读者出现
    sysopen(FIFO, $fpath, O_WRONLY)
        || die "can't write $fpath: $!";
    print FIFO "John Smith (smith\@host.org)\n", `fortune -s`;
    close FIFO;
    select(undef, undef, undef, 0.2); # 睡眠1/5秒
}

```

close之后的短暂睡眠是必要的，这样可以让读者有机会读刚才所写的内容。如果立即再次循环，在读者读完刚才发送的数据之前就再次打开FIFO，这样一来，将不会看到end-of-file（文件末尾），因为又有了一个写者。我们将如此循环，直到在某次循环迭代中写者稍微慢一点，此时读者才能最终看到久违的end-of-file（而且我们还要担心竞态条件，不是吗）。

## System V IPC

所有人都不喜欢System V IPC。它比纸带机还慢，会不知不觉地建立一些与文件系统完全无关的小命名空间，而且使用人们讨厌的数字来命名对象，另外还经常“忘事”。你的系统管理员不得不经常做这种搜索-撤销的任务，先用`ipcs(1)`找到那些丢失的SysV IPC对象，再用`ipcrm(1)`将其终止（希望系统还没有把内存耗尽）。

尽管这些很费事，但古老的SysV IPC还是有一些有效的用法。有三种IPC对象：共享内存、信号量和消息。对于消息传递，如今更合适的机制是套接字，而且套接字有更好的可移植性。对于简单的信号量，现在往往会使用文件系统。而对于共享内存，嗯，这里有一个问题。更好的方法是使用更现代的`mmap(2)`系统调用（如果系统提供支持），不过实现的质量会因系统的不同而不同。另外还要特别当心，不要让Perl从`mmap(2)`放置字符串的地方为字符串重新分配空间。

`File::Map` CPAN模块会让这个工作容易得多。处理时仍然要很小心，不过如果你搞砸了，它只是警告你而不会因内存段冲突转存内核。

下面给出一个小程序，这里展示了一些兄弟进程对一个共享内存缓冲区的受控访问。SysV IPC对象也可以由同一台计算机上的无关进程共享，不过必须确定它们以何种方式相互找到对方。为了保证安全访问，我们将为每一个共享内存创建一个信号量<sup>注10</sup>。

每次想要从共享内存获取一个值，或者要在其中放一个新值时，必须先经过信号量。这可

---

注10：更现实的做法是为每一个共享内存创建一对信号量，一个用于读，另一个用于写。实际上，这正是CPAN上`IPC::Shareable`模块的做法。不过这里我们想更简单些。但必须承认，使用一对信号量可以利用SysV IPC唯一的优点：可以将整个信号量集合作为一个单元来完成原子操作，这一点在某些情况下非常有用。



能很麻烦，所以我们把这个访问包装在一个对象类中。IPC::Shareable则更进一步，将其对象类包装在一个tie接口中。

这个程序会一直运行，直到你用一个Control-C或类似的序列将其中断：

```
#!/usr/bin/perl -w
use v5.6.0; # or better
use strict;
use sigtrap qw(die INT TERM HUP QUIT);
my $PROGENY = shift(@ARGV) || 3;
eval { main() }; # 参考下面的DESTROY来了解为什么
die if $@ && $@ !~ /^Caught a SIG/;
print "\nDone.\n";
exit;

sub main {
    my $mem = ShMem->alloc("Original Creation at " . localtime);
    my(@kids, $child);
    $SIG{CHLD} = "IGNORE";
    for (my $unborn = $PROGENY; $unborn > 0; $unborn--) {
        if ($child = fork) {
            print "$$ begat $child\n";
            next;
        }
        die "cannot fork: $!" unless defined $child;
        eval {
            while (1) {
                $mem->lock();
                $mem->poke("$$ " . localtime)
                    unless $mem->peek =~ /^$$\b/o;
                $mem->unlock();
            }
        };
        die if $@ && $@ !~ /^Caught a SIG/;
        exit; # 子进程终止
    }
    while (1) {
        print "Buffer is ", $mem->get, "\n";
        sleep 1;
    }
}
```

下面给出程序使用的ShMem包。可以把它附在程序的末尾，也可以放在单独的文件中（末尾加一个“1;”），并在主程序中通过require加载（它使用的两个IPC模块可以在标准Perl版本中找到）。

```
package ShMem;
use IPC::SysV qw(IPC_PRIVATE IPC_RMID IPC_CREAT S_IRWXU);
use IPC::Semaphore;
sub MAXBUF() { 2000 }

sub alloc { # 构造方法
```

```

my $class = shift();
my $value = @_ ? shift() : "";

my $key = shmget(IPC_PRIVATE, MAXBUF, S_IRWXU) || die "shmget: $!";
my $sem = IPC::Semaphore->new(IPC_PRIVATE, 1, S_IRWXU | IPC_CREAT)
    or die "IPC::Semaphore->new: $!";
$sem->setval(0,1) or die "sem setval: $!";
my $self = bless {
    OWNER => $$,
    SHMKEY => $key,
    SEMA => $sem,
} => $class;

$self->put($value);
return $self;
}

```

下面给出获取（fetch）和存储（store）方法。get和put方法会锁定缓冲区，而peek和poke不会，所以只有当对象手动锁定时才能使用后面这两个方法，如果希望获取原来的值，然后存回修改后的新值，所有这些操作都要有同一个锁，这种情况下就必须手动锁定。这个示例程序在while（1）循环中完成这些工作。整个事务必须在同一个锁下进行，否则测试和设置就不是原子操作，可能会出问题。

```

sub get {
    my $self = shift();
    $self->lock;
    my $value = $self->peek(@_);
    $self->unlock;
    return $value;
}
sub peek {
    my $self = shift();
    shmread($self->{SHMKEY}, my $buff=q(), 0, MAXBUF) || die "shmread: $!";
    substr($buff, index($buff, "\0")) = q();
    return $buff;
}
sub put {
    my $self = shift();
    $self->lock;
    $self->poke(@_);
    $self->unlock;
}
sub poke {
    my($self,$msg) = @_;
    shmwrite($self->{SHMKEY}, $msg, 0, MAXBUF) || die "shmwrite: $!";
}
sub lock {
    my $self = shift();
    $self->{SEMA}->op(0,-1,0) || die "semop: $!";
}
sub unlock {
    my $self = shift();
    $self->{SEMA}->op(0,1,0) || die "semop: $!";
}

```

最后，这个类需要一个析构函数。对象撤销时，我们可以手动地释放共享内存以及对象中存储的信号量。否则，它们会比其创建者寿命还要长，这样一来，我们就不得不求助于 *ipcs* 和 *ipcrm*（或系统管理员）将其清除。正是因为这个原因，我们在主程序中设计了包装器将信号转换为异常：异常会导致运行析构函数，并释放 SysV IPC 对象，这就不再需要麻烦系统管理员了。

```
sub DESTROY {  
    my $self = shift();  
    return unless $self->{OWNER} == $$; # 避免双重释放  
    shmctl($self->{SHMKEY}, IPC_RMID, 0) || warn "shmctl RMID: $!";  
    $self->{SEMA}->remove() || warn "sema->remove: $!";  
}
```

## 套接字

前面讨论的所有IPC机制都有一个严重的限制：它们都是为同一台计算机上运行的进程之间建立通信而设计的（尽管有时文件可以通过类似NFS等机制在不同机器之间共享，不过不幸的是，很多NFS实现中锁定会失败，这就失去了并发访问的大部分乐趣）。对于通用目的的网络通信，最好的办法是使用套接字。尽管套接字是在BSD下发明的，不过很快传播到其他Unix系统，如今几乎每一个可用的操作系统上都可以找到一个套接字接口。如果你的机器上没有套接字，要想使用互联网会有很大难度。

利用套接字，可以建立虚电路（作为TCP流），也可以建立数据报（作为UDP数据包）。套接字还可以做得更多，这取决于你的系统。不过，最常见的套接字编程都使用基于Internet域套接字的TCP，这也是我们将要介绍的内容。这种套接字可以提供可靠的连接，它的工作有点类似于双向管道，但不限于同一台主机。互联网上的两大流行应用email和Web浏览几乎都完全依赖于TCP套接字。

尽管你可能没有意识到，不过UDP也在大量使用。每次你的机器想要在互联网上查找一个网站时，它会向DNS服务器发送UDP数据包，来请求具体的IP地址。发送和接收数据报时，你自己也可以使用UDP。数据报比TCP连接更经济，因为数据报不是面向连接的；也就是说，它们不太像打电话，而更像在邮箱里投递信件。不过UDP缺少TCP的可靠性，这使它更适合那些不太关心安全性的场合，比如你不在乎丢掉和多出一、两个数据包，或者不在意一、两个数据包被篡改。如果你知道一个更高层协议会强制某种程度的冗余性或故障弱化（DNS就是这样），也很合适使用UDP。

还有另外一些套接字选择，不过不太常用。可以使用Unix域套接字，但它们只用于本地通信。很多系统还支持其他非IP协议。毫无疑问，肯定有些人会对这些套接字感兴趣，不过这里不做太多介绍。

处理套接字的Perl函数与C中相应的系统调用同名，不过它们的参数不同，这有两个原因：首先，Perl文件句柄的工作方式与C文件描述符不同；其次，Perl已经知道其字符串的



长度，所以不需要传递这个信息。可以参见第27章了解与套接字有关的各个系统调用的详细内容。

老的Perl套接字代码存在一个问题，人们往往使用硬编码的值作为传入套接字函数的常量，这会破坏可移植性。与大多数系统调用一样，与套接字有关的调用如果失败，会安静而礼貌地返回`undef`，而不是产生一个异常。因此检查这些函数的返回值至关重要，因为即使你传入垃圾，它们也不会为此大声叫嚷。如果你看到类似这样的代码（如显式地设置`$AF_INET = 2`），就应该知道有大麻烦了。一种更好的方法是使用`Socket`模块，甚至可以使用更友好的`IO::Socket`模块，这两个模块都是标准模块。这些模块提供了建立客户端和服务端时需要的很多常量和辅助函数。要想尽可能成功，套接字程序的开头应该类似这样（不要忘记为服务器增加-T污染检查开关）：

```
#!/usr/bin/perl
use v5.14;
use warnings;
use autodie;

#对于IPv6，可以使用CPAN的IO::Socket::IP
use IO::Socket;
```

正如别处所介绍的，Perl要依赖C库来完成它的很多系统行为，而且并非所有系统都支持所有不同类型的套接字。最安全的做法可能是只使用常规的TCP和UDP套接字操作。例如，如果你希望代码能够移植到其他系统（甚至是你没有想到过的系统），就不要假设那个系统会支持可靠的顺序报文协议，也不要期望通过一个本地Unix域套接字在不相关的进程之间传递打开的文件描述符（没错，在很多Unix机器上确实可以做到这一点，参见你本机的`recvmsg(2)`手册页）。

如果你只想使用一个标准的Internet服务，如邮件、新闻组、域名服务、FTP、Telnet、Web等等，不用从零开始，可以使用已有的相应CPAN模块。为此已经建立了一些预打包的模块，包括`Net::SMTP`（或`Mail::Mailer`）、`Net::NNTP`、`Net::DNS`、`Net::FTP`、`Net::Telnet`和各种与HTTP相关的模块。`libnet`和`libwww`模块集由很多单独的网络通信模块组成。

在下一节中，我们将给出几个示例客户端和服务端程序，不过不会对所使用的各个函数做太多解释，否则会与第27章的描述重复。

## 网络客户端

如果希望在可能不同的机器之间建立可靠的客户-服务器通信，要使用Internet域套接字。

要创建一个TCP客户端与某个位置的服务器连接，通常最容易的做法是使用标准`IO::Socket::INET`模块：

```
#!/usr/bin/env perl
use v5.14;
use warnings;
use autodie;
use IO::Socket::INET;

my $remote_host = "localhost";      # 替换为实际的远程主机
my $remote_port = "daytime";       # 替换为服务器名或端口号

my $socket = IO::Socket::INET->new(
    PeerAddr => $remote_host,
    PeerPort => $remote_port,
    Type => SOCK_STREAM,
);

# 通过套接字发送数据，如CRLF之类
# daytime不接收输入，而是在其他服务器上使用
print $socket "Why don't you call me anymore?\r\n";

# 读取远程响应
my $answer = <$socket> =~ s/\R\z//r;

say "Got answer: $answer";

# 完成工作时终止连接
close($socket);
```

如果只需指定要连接的主机和端口，所有其他字段可以使用默认值，那么完全可以使用这个调用的一种简写形式：

```
$socket = IO::Socket::INET->new("www.yahoo.com:80")
or die "Couldn't connect to port 80 of yahoo: $!";
```

对于IPv6，如果能够从CPAN得到IO::Socket::IP模块，使用这个模块是最容易的。如果有比v5.14更新的Perl版本，可能你的系统上已经有这个模块了。一旦找到这个模块，你要做的只是把以上代码中的类名由IO::Socket::INET改为IO::Socket::IP，这个代码同样可以用于IPv6。这个类有一个额外的sockdomain方法，可以查看你的IP版本：

```
#!/usr/bin/env perl
use v5.14;
use warnings;
use autodie;
use IO::Socket::IP;

my $remote_host = "localhost";
my $remote_port = "daytime";

my $socket = IO::Socket::IP->new(
    PeerAddr => $remote_host,
    PeerPort => $remote_port,
    Type => SOCK_STREAM,
);
```

```

my $familyname = ( $socket->sockdomain == AF_INET6 ) ? "IPv6" :
                  ( $socket->sockdomain == AF_INET ) ? "IPv4" :
                  "unknown";

say "Connected to $remote_host:$remote_port via ", $familyname;

# 通过套接字发送数据，如CRLF
print $socket "Why don't you call me anymore?\r\n";

# 读取远程响应
my $answer = <$socket> =~ s/\R/z//r;

say "Got answer: $answer";

# 完成工作时终止连接
close($socket);

```

使用基本Socket模块连接：

```

use v5.14;
use warnings;
use autodie;
use Socket;
my $remote_host = "localhost";
my $remote_port = 13; # daytime服务端口

socket(my $socket, PF_INET, SOCK_STREAM, getprotobyname("tcp"));
my $internet_addr = inet_aton($remote_host);
my $paddr = sockaddr_in($remote_port, $internet_addr);

connect($socket, $paddr);
$socket->autoflush(1);

print $socket "Why don't you call me anymore?\r\n";
my $answer = <$socket> =~ s/\R/z//r;

say "Answer was: ", $answer;

```

在v5.14中，可以结合标准Socket模块使用IPv6，不过函数调用和API与以上所示稍有区别，上面的代码只适用于IPv4。有关的详细内容参见Socket手册页。

如果希望只关闭本地端的连接，使远端得到文件末尾（end-of-file），不过仍然能读取来自服务器的数据，为此可以使用shutdown系统调用来完成这种“半关闭”：

```

# 对服务器没有更多写操作
shutdown(Server, 1); # v5.6中的Socket::SHUT_WR常量

```

## 网络服务器

下面是与以上客户端对应的服务器。利用IO::Socket::INET类很容易实现：

```

use IO::Socket::INET;

```



```

$server = IO::Socket::INET->new(LocalPort => $server_port,
                                Type => SOCK_STREAM,
                                Reuse => 1,
                                Listen => 10 ) # 或SOMAXCONN
    || die "Couldn't be a tcp server on port $server_port: $!\n";

while ($client = $server->accept()) {
    # $client是新连接
}

close($server);

```

还可以使用底层Socket模块来实现：

```

#!/usr/bin/env perl

use v5.14;
use warnings;
use autodie;
use Socket;

my $server_port = 12345; # 选择一个数字

# 建立套接字
socket(my $server, PF_INET, SOCK_STREAM, getprotobyname("tcp"));

# 从而能很快地重启我们的服务器
setsockopt($server, SOL_SOCKET, SO_REUSEADDR, 1);

# 建立套接字地址
my $own_addr = sockaddr_in($server_port, INADDR_ANY);
bind($server, $own_addr);

# 为到来的连接建立一个队列
listen($server, SOMAXCONN);

# 接受并处理连接
while (accept(my $client, $server)) {
    # 对$client中的新客户连接做些处理
} continue {
    close $client;
}

close($server);

```

客户端不需要绑定到任何地址，不过服务器必须绑定到某个地址。我们指定其地址为 `INADDR_ANY`，这表示客户可以从任何可用的网络接口连接。如果你希望一直使用某个特定的接口（如网关或防火墙机器），就要使用该接口的真实地址（客户端也可以这样做，但是很少需要这么做）。

如果你想知道哪个机器与你连接，可以在客户连接上调用 `getpeername`。这会返回一个IP地址，你必须自行将它转换为一个名字（如果你做得到的话）：

```

use Socket;
$other_end = getpeername($client)
    || die "Couldn't identify other end: $!\n";
($port, $iaddr) = unpack_sockaddr_in($other_end);
$actual_ip = inet_ntoa($iaddr);
$claimed_hostname = gethostbyaddr($iaddr, AF_INET);

```

这种方法很容易被欺骗，因为该IP地址的所有者可以建立他的逆向表，设置为他想设置的任何名字。为了进一步确认，再从另一个方向转换为IP地址：

```

@name_lookup = gethostbyname($claimed_hostname)
    || die "Could not reverse $claimed_hostname: $!\n";
@resolved_ips = map { inet_ntoa($_) } @name_lookup[ 4 .. $#name_lookup ];
$might_spoof = !grep { $actual_ip eq $_ } @resolved_ips;

```

一旦客户端与服务器连接，服务器可以对这个客户端句柄完成I/O操作。不过，由于服务器在投入地工作，它无法为来自其他客户端的更多请求提供服务。为了避免一次只锁定一个客户端，很多服务器会立即用fork派生自己的一个克隆来处理每一个到来的连接（另外一些服务器可能会预先用fork创建克隆，或者使用select系统调用在多个客户端之间复用I/O）。

```

REQUEST:
while (accept(my $client => $server)) {
    if ($kidpid = fork) {
        close $client; # 父进程关闭不用的句柄
        next REQUEST;
    }
    defined($kidpid) || die "cannot fork: $!";

    close $server; # 子进程关闭不用的句柄

    $client->autoflush(1);

    # 每个连接的子进程代码用Client句柄完成I/O
    $input = <$client>;
    print $client "output\n"; # 或STDOUT，同样的结果

    open(STDIN, "<&", $client)      || die "can't dup client: $!";
    open(STDOUT, ">&", $client)     || die "can't dup client: $!";
    open(STDERR, ">&", $client)    || die "can't dup client: $!";

    # 运行计算器，只是一个例子
    system("bc -l");               # 或者你喜欢的任何东西，只要
                                    # 它没有shell转义！
    print "done\n";                # 仍输出到客户端
    close $client;
    exit; # 不要让子进程返回accept!
}

```

这个服务器会为每一个到来的请求用fork克隆一个子进程。这样一来，它可以同时处理多个请求（只要你能创建多个进程）。你可能希望对此加以限制。即使没有用fork创建子进

程，`listen`也会允许最多`SOMAXCONN`个（通常5个或更多）到来的连接。每个连接会使用一些资源，不过没有进程使用的资源多。为服务器创建子进程时要注意清理其过期的子进程（在Unix里称为“僵尸进程”），否则它们会很快填满你的进程表。前面“信号”一节中讨论的`REAPER`代码可以处理这个问题，或者可以赋值`$SIG{CHLD} = "IGNORE"`。

运行另一个命令前，我们把标准输入和输出（以及错误输出）连接到客户端连接。这样一来，所有从`STDIN`读和写至`STDOUT`的命令也可以与远程机器通信。如果没有重新赋值，命令就无法找到客户端句柄。毕竟，默认情况下到达`exec`边界时（即执行结束时）客户端句柄就会关闭。

写网络服务器时，强烈建议使用`-T`开关来启用污染检查（即使你没有运行`setuid`或`setgid`）。这对于服务器和代表其他进程运行的程序（如所有CGI脚本）都很有好处，因为这样可以减少外部的人破坏系统的机会。有关的更多内容参见第20章“处理不安全的数据”一节。

写Internet程序时还要注意一点：很多协议指定行结束符应当是`CRLF`，这可能会以不同的方式指定：`"\r\n"`<sup>注11</sup>、`"\015\12"`或`"\xd\xa"`，或者甚至可以是`chr(13).chr(10)`。很多Internet程序实际上接受一个简单的`"\012"`作为行结束符，不过这是因为Internet程序通常对接收的内容很宽松，而对它们发出的内容很严格（如果能让大家也这样就好了）。

## 消息传递

前面我们提到过，UDP通信的开销低得多，不过不提供可靠性，因为它不能保证消息会以正确的顺序到达，甚至不能保证消息肯定会到达。人们常把UDP称为不可靠数据报协议（Unreliable Datagram Protocol）。

不过，UDP具有TCP所没有的一些优点，包括能够同时广播或组播到大量目标主机（通常在你的本地子网中）。如果开始关心可靠性，而且开始在消息系统中增加检查，这说明你可能应该使用TCP。没错，建立和解除TCP连接的开销更大，不过如果把这个开销分摊到多个消息（或者一个很长的消息），这也是值得的。

下面给出一个UDP程序的例子。它要与命令行中指定机器的UDP时间端口通信，如果没有提供任何参数，则与使用统一广播地址找到的所有人通信<sup>注12</sup>。并不是所有机器都启用了时间服务器，特别是跨防火墙边界时，不过启用时间服务器的机器确实会向你发回一个4字节的整数（按网络字节顺序打包），这个整数表示该机器所认为的时间。不过，所返回的时间用1900年以来的秒数表示。你必须减去1900~1970年之间的秒数，这个时间才能提供给`localtime`或`gmtime`转换函数。

---

注11：很久以前的Unix前身Macs除外，不过已经没有人用这个系统了。

注12：如果这样不行，可以运行`ifconfig -a`来查找适当的本地广播地址。



```

#!/usr/bin/perl
# clockdrift - 将其他系统的时钟与这个系统的时钟比较
#              没有参数，广播到所有监听者。
#              等1.5秒，等待回应。

use v5.14;
use warnings;
use strict;
use Socket;

unshift(@ARGV, inet_ntoa(INADDR_BROADCAST))
    unless @ARGV;

socket(my $msgsock, PF_INET, SOCK_DGRAM, getprotobyname("udp"))
    || die "socket: $!";

# 有些怪异的机器需要这一点。不会影响其他机器
setsockopt($msgsock, SOL_SOCKET, SO_BROADCAST, 1)
    || die "setsockopt: $!";

my $portno = getservbyname("time", "udp")
    || die "no udp time port";

for my $target (@ARGV) {
    print "Sending to $target:$portno\n";
    my $destpaddr = sockaddr_in($portno, inet_aton($target));
    send($msgsock, "x", 0, $destpaddr)
        || die "send: $!";
}

# 时间服务返回一个32位时间，用自1900年以来的秒数表示
my $FROM_1900_TO_EPOCH = 2_208_988_800;
my $time_fmt = "N"; # 用二进制格式
my $time_len = length(pack($time_fmt, 1)); # 任何数字都可以。

my $inmask = q(); # 存储select fileno位的字符串。
vec($inmask, fileno($msgsock), 1) = 1;
# 等1.5秒，等待输入出现
while (select(my $outmask = $inmask, undef, undef, 0.5)) {
    defined(my $srcpaddr = recv($msgsock, my $bintime, $time_len, 0))
        || die "recv: $!";
    my($port, $ipaddr) = sockaddr_in($srcpaddr);
    my $sendhost = sprintf "%s [%s]",
        gethostbyaddr($ipaddr, AF_INET) || "UNKNOWN",
        inet_ntoa($ipaddr);
    my $delta = unpack($time_fmt, $bintime) -
        $FROM_1900_TO_EPOCH - time();
    print "Clock on $sendhost is $delta seconds ahead of this one.\n";
}

```

# 编译

如果你来这里找一个Perl编译器，可能会惊讶地发现已经有这样的一个编译器，perl程序（通常是`/usr/bin/perl`）已经包含一个Perl编译器。这可能不是你想要的编译器，如果是这样，你会高兴地发现我们还提供了一些代码生成器（code generators），一些好心的人把它们称为“编译器”，这正是本章要讨论的内容。不过，首先来说说我们所认为的编译器是什么。不可避免地，这一章会有一些比较底层的细节，有些人可能对此感兴趣，不过也有一些人可能毫无兴趣。如果你属于后一类，可以把这当做一个锻炼快速阅读能力的机会。

假设你是一个指挥家，已经拿到了一个大型管弦乐队的排练乐谱。收到乐谱时，你发现有几十本小册子，乐队每个成员人手一册，其中只有每个人自己的那一部分。不过，奇怪的是，这里没有涵盖所有部分的主乐谱。更奇怪的是，各部分乐谱都使用英语书写，而没有使用标准的音乐符号。在准备节目之前，甚至在乐队排练之前，首先必须把这些“文字描述”翻译成用音符和小节线表示的正常乐谱。然后还需要把各个部分整理为一个完整的大乐谱，以便对整个节目有一个整体认识。

类似地，你把Perl脚本的源代码交给perl执行时，对于计算机来说，这些源代码没有任何意义，就像把用英语写的交响乐谱交给音乐家一样。程序运行前，Perl需要把这些类似英语的文字编译<sup>注1</sup>为一种特殊的符号表示。不过还不会运行你的程序，因为编译器只完成编译工作。就像指挥的乐谱一样，即使你的程序已经转换为一种适合解释的指令格式，不过还需要一个“主体”来具体解释这些指令。

---

注1： 或翻译、转换、改变、变换或者变形。

# Perl程序的生命周期

可以把Perl程序的生命周期划分为4个不同的阶段，这些阶段还可以分别细分为更小的子阶段。第一个和最后一个阶段最有意思，中间两个阶段是可选的。这些阶段如图16-1所示。

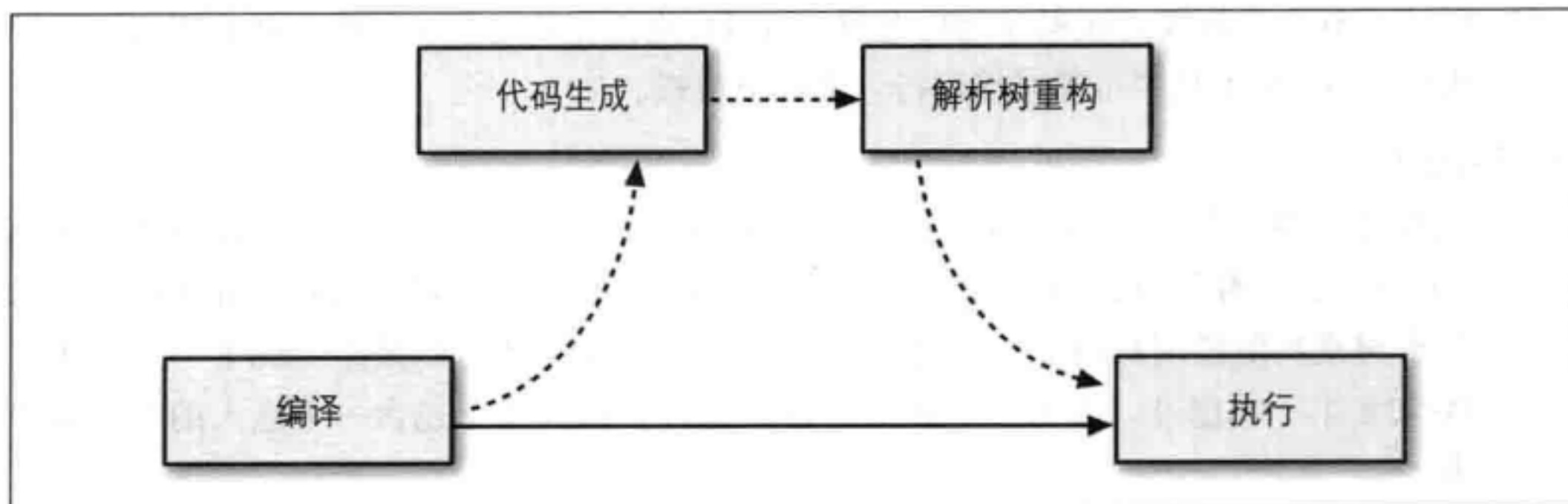


图16-1：Perl程序的生命周期

## 1. 编译阶段

在第1阶段，即编译阶段（compile phase），Perl编译器将程序转换为一种称为解析树（parse tree）的数据结构。除了标准的解析技术外，Perl还采用了一种更强大的技术：使用BEGIN块来指导进一步的编译。一旦解析到BEGIN块，会把它交给解释器运行，解释器将以FIFO（先进先出）的顺序来运行这些BEGIN块，这也包括所有use和no声明（实际上use和no声明就是伪装的BEGIN块）。一旦编译单元完成编译，会执行UNITCHECK块，这些UNITCHECK块可以用于完成每个单元的初始化。所有CHECK、INIT和END块由编译器调度延迟执行。

词法作用域变量声明会加标记，但并不执行其赋值。所有eval BLOCK、s///e构造和非内插正则表达式都在这个阶段编译，另外常量表达式将完成预计算。编译器的工作就完成了，除非以后再来调用它。在这个阶段的最后，再次调用解释器以LIFO（后进先出）的顺序执行所有已调度的CHECK块。是否有CHECK块决定了下一步进入第2阶段还是直接跳至第4阶段。

## 2. 代码生成阶段（可选）

这个阶段是可选的，CHECK块由代码生成器安装，所以如果你显式地使用了某个代码生成器（见本章后面“代码生成器”一节），就会出现这个阶段。代码生成器将已编译（但尚未运行）的程序转换为C源代码或串行Perl字节码（bytecodes），字节码是表示内部Perl指令的一个值序列。如果选择生成C源代码，最后会生成一个原生机器语言文件，称为可执行映像（executable image）<sup>注2</sup>。此时，你的程序进入“假

注2： 原来的脚本也是一个可执行文件（executable file），不过不是机器语言，所以不能把它叫做映像。一些文件之所以被称为映像文件，是因为它是机器代码的一个原文副本（CPU知道如何执行这些机器代码）。



死”状态。如果建立了一个可执行映像，可以直接进入第4阶段，否则需要在第3阶段重新构造“冻干”的字节码。

### 3. 解析树重构阶段（可选）

要让程序复活，必须重构其解析树。只有当发生了代码生成而且你选择生成字节码时才会有这个阶段。在程序运行之前，Perl必须首先从这个字节码序列重新构造其解析树。Perl并不从字节码直接运行，那样会很慢。

### 4. 执行阶段

最后终于到了你一直等待的阶段：运行你的程序。因此，这也称为运行阶段（run phase）。解释器得到解析树（可以直接从编译器得到，或者从代码生成和接下来的解析树重构阶段间接得到），并执行这个解析树（或者，如果你生成了一个可执行映像文件，它也可以作为一个独立的程序运行，因为其中包含一个嵌入的Perl解释器）。

这个阶段开始时，在主程序即将运行之前，首先所有已调度的INIT块会以FIFO顺序执行。然后运行你的主程序。遇到`eval STRING`、`do FILE`或`require`语句、`s///ee`构造或者有一个内插变量（包含一个合法代码断言）的模式匹配时，解释器可以根据需要再次调用编译器。

主程序执行结束后，所有延迟的END块终于得以执行，这一次要以LIFO顺序执行。第一个看到的END块将最后执行，然后就大功告成了。只有两种情况有所例外，如果你使用了`exec`，或者你的进程被某个未捕获的灾难性错误终止，此时会跳过END块。普通的异常并不认为是灾难性的。

下面我们将更详细地讨论所有这些阶段，不过介绍的顺序不同。

## 编译代码

Perl总是处于两种操作模式之一：要么正在编译你的程序，要么正在执行程序，这二者绝对不会同时进行。在这本书中，我们提到过某些事件在编译时发生，或者我们会说“Perl编译器做某件事”。另外，我们也提到某些事情在运行时发生，或者“Perl解释器做某件事”。尽管你可能把编译器和解释器都认为是“Perl”，不过要知道Perl在给定时刻具体承担其中的哪一个角色，这对于理解很多事情为什么会发生是至关重要的。这两个角色Perl都会实现：首先是编译器，然后是解释器（还可能还有其他角色；perl也是一个优化器和代码生成器。有时，它甚至还是一个魔术师，不过是个很有趣的魔术师）。

编译阶段和编译时之间以及运行阶段与运行时之间是有区别的，理解这个区别也很重要。典型的Perl程序有一个编译阶段，然后是一个运行阶段。“阶段”是一个大范围的概念，而编译时和运行时是小范围的概念。一个给定的编译阶段会完成大部分编译时工作，不过

也可能通过BEGIN块完成一些运行时工作。一个给定的运行阶段主要完成运行时工作，不过也可能通过类似eval *STRING*等操作符完成编译时工作。

在典型的事件过程中，Perl编译器在执行开始前先读取整个程序源代码。此时Perl要解释声明、语句和表达式，确保它们在语法上是正确的<sup>注3</sup>。如果发现一个语法错误，编译器会尝试从这个错误恢复回来，从而能报告源代码中后面的其他错误。有时这是可以恢复的，不过有时无法恢复，语法错误有一个让人讨厌的倾向，总是会触发一大堆的假警报。Perl在遇到10个错误之后就会无奈地异常终止。

除了处理BEGIN块的解释器外，编译器还会利用3个代理过程来处理程序。词法分析器（lexer）扫描程序中的每一个最小的意义单位。有时这些最小单位称为词法单位（lexeme），不过在关于编程语言的文章中可能把它们更多的称为词法标记（token）。词法分析器有时也称为标记分析器（tokenizer）或扫描器（scanner），而它做的工作有时则称为词法分析或标记分析。解析器（parser）则尝试根据Perl语言的文法将这些token组装为更大的构造，如表达式和语句，以了解这些token组的含义。优化器（optimizer）重新组织这些较大的token组，将它们缩减为更高效的序列。它会仔细地选择优化方法，不会浪费时间纠结于边缘优化，因为作为一个装入即运行的编译器，Perl编译器必须保证非常快速地运行。

这些并非在单独的阶段中分别完成，而会同时发生，词法分析器、解析器和优化器之间有大量交互。词法分析器有时需要来自解析器的提示，才能知道它要查看哪些可能的token类型（奇怪的是，词法分析器并不理解词法作用域，因为词法作用域中的“词法”有另一层含义）。优化器还需要跟踪解析器的工作，因为有些优化在解析到某个点之前可能不会发生，如完成一个表达式、语句、块或子例程。

Perl编译器会同时完成所有这些事情，而不是一件一件地先后完成，你可能认为这样很奇怪。不过，你听读自然语言时，理解过程实际上也差不多。你不会等到读完一章之后再确定第一句话是什么意思。请看表16-1所列的对应关系。

表16-1：计算机语言和自然语言中的对应项

计算机语言	自然语言
字符	字母
标记（Token）	词素
项	单词
表达式	短语

注3： 这里没有类似BNF的形式化语法图，不过你可以仔细研究Perl源代码树中的perly.y文件，其中包含Perl使用的yacc(1)文法。建议你少用词法分析器（lexer），大家都已经知道它存在一些问题。



表16-1：计算机语言和自然语言中的对应项（续）

计算机语言	自然语言
语句	句子
块	段落
文件	章
程序	故事

假设解析顺利进行，编译器认为你的输入是一个合法的“故事”，噢，应该说是合法的程序。如果运行程序时使用了`-c`开关，它会打印一个“syntax OK”（语法正确）消息，然后退出。否则，编译器会把它的解析成果传递到其他代理过程。这些“成果”采用解析树的形式。树上的每个果实（或通常称为“节点”）表示Perl的一个内部操作码（opcode），而树上的分支表示树的历史增长模式。这些节点最后会一个接一个地线性地串在一起，指示运行时系统访问这些节点的执行顺序。

操作码是Perl的可执行指令的最小单位。你可能把类似`$a = -($b + $c)`的表达式看作是一个语句，不过Perl会认为这是6个不同的操作码。如果采用一种简化形式显示，这个表达式的解析树如图16-2所示。数字表示Perl运行时系统最后遵循的访问顺序。

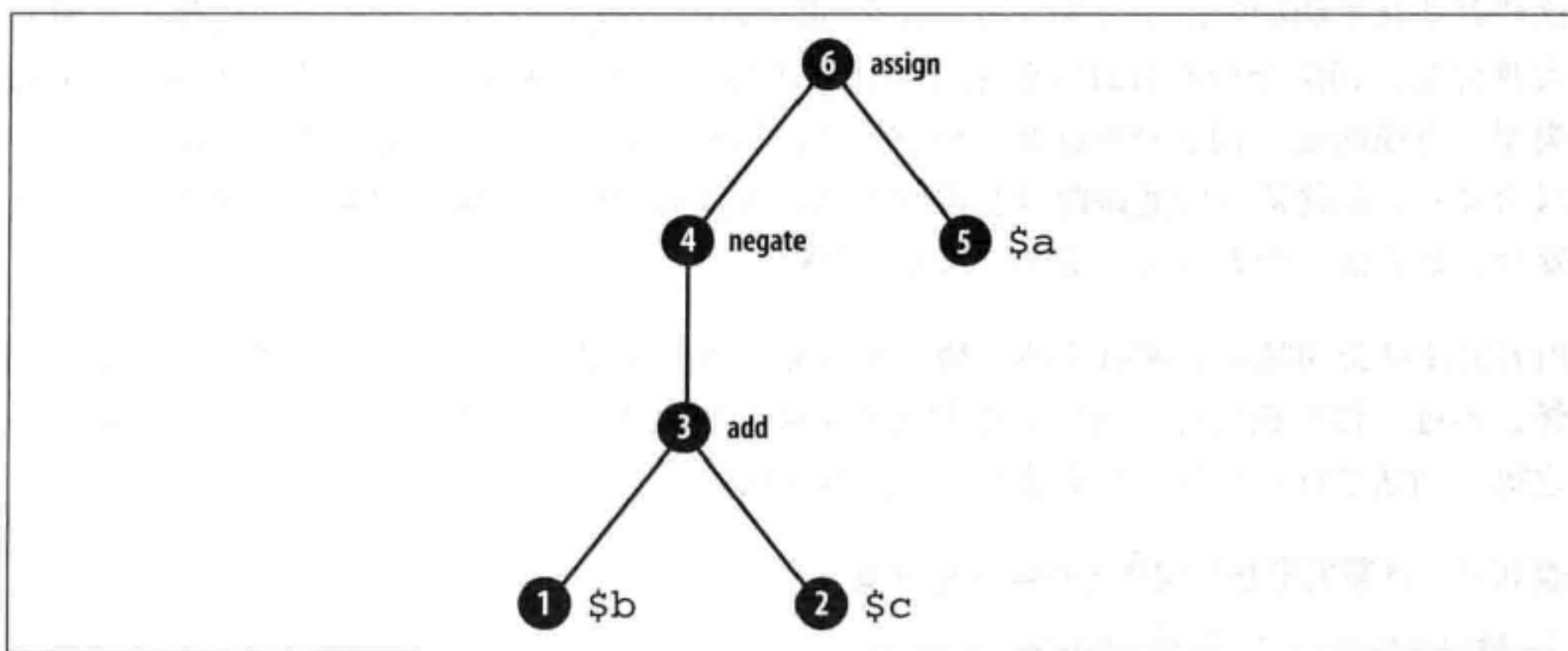


图16-2：`$a = -($b + $c)`的操作码访问顺序

与有些人想象的不一样，Perl不是一个一趟式（One-pass）编译器（“一趟式”编译器最大的特点是让计算机很容易，而让程序员很困难）。它是一个多趟式的优化编译器，包含至少3趟不同的逻辑处理，它们实际上会交替进行。解析树构造过程中，编译器会对解析树上上下下地反复处理，第1趟和第2趟交替运行，子例程或文件完全解析时会发生第3趟处理。各趟处理过程如下：



## 第1趟: 自底向上解析

在这一趟处理中, 由yacc(1)解析器使用从底层词法分析器得到的token (这本身也可以认为是另一趟逻辑处理) 构建解析树。自底向上意味着解析器先知道树的树叶, 然后才能知道其分支和树根。它确实是自下而上来确定图16-2所示的节点, 因为我们是按照计算机科学家 (和语言学家) 的特殊方式来绘制这棵树, 把树根画在最上面。

构造各个操作码节点时, 会对每个操作码完成合理性检查, 要验证语义是否正确, 如调用内置函数时参数的个数和类型必须正确。随着树的各个部分逐步成形, 优化器会考虑对整个子树可以应用哪些变换。例如, 假设一个函数接受指定数目的参数, 一旦知道要为此函数输入一个值列表, 优化器可以不考虑为变参函数记录参数个数的操作码。本节后面还会介绍一个更重要的优化, 称为常量叠算 (constant folding)。

这一趟还会建立节点访问顺序, 这个顺序在后面的执行中将会用到, 这是一个很不错的技巧, 因为最先访问的节点几乎不可能是顶节点。编译器会建立一个临时的操作码环, 顶节点指向要访问的第一个操作码。顶层操作码结合为更大的操作码时, 将断开这个操作码环, 建立一个更大的环, 并有新的顶节点。当起始操作码结合到另外某个结构中时 (如一个子例程描述符), 这个环最终完全断开。子例程调用者仍能找到第一个操作码, 尽管它在树的最底层, 如图16-2所示。解释器没有必要在解析树中一直向下回溯来确定从哪里开始。

## 第2趟: 自顶向下优化器

一个人如果只读一小段Perl代码 (或者英文文章), 倘若不检查前后的词法元素, 就无法确定上下文。有时, 在得到更多信息之前, 你无法确定到底发生了什么。不过, 不用担心, 因为并不只有你是这样, 编译器也一样。在这一趟中, 编译器在刚建立的子树中下行来应用局部优化, 其中最值得注意的是上下文传播 (context propagation)。编译器将标记下级节点, 指示它采用当前节点所施加的适当的上下文 (void、标量、列表、引用或左值上下文)。不再需要的操作码会清空但并不删除, 因为现在重新构造执行顺序已经为时过晚。我们要靠第3趟将它们从第1趟确定的临时执行顺序删除。

## 第3趟: 窥视孔优化器

某些代码单元有自己的存储空间, 可以在其中保存词法作用域变量 (按照Perl的术语来讲, 这种空间称为便签簿 (scratchpad))。这些单元包括eval STRING、子例程和整个文件。更重要的是, 从优化器的角度来讲, 它们分别有自己的入口点, 这说明尽管我们知道从这里开始之后的执行顺序, 但是无法知道之前发生了什么, 因为这个构造可以从任何位置调用。所以解析这样一些单元时, Perl会对这个代码运行一个窥视孔优化器。之前的两趟都沿着解析树的分支结构遍历, 与它们不同, 这一趟会按线性执行顺序遍历代码, 因为在从解析器 “切除” 这个操作码列表之前, 这实

际上是最后一个解析机会。大多数优化已经在前两趟完成，不过有些优化还没有完成。

这里会发生很多晚期优化，包括通过忽略清空的操作码，以及识别什么时候可以将各种操作码排列缩减为更简单的组合，从而“拼接”出最终的执行顺序。识别字符串连接就是一个很重要的优化，因为你肯定不希望每次在末尾增加一点东西时都不得不再复制字符串。这一趟并不只是优化，它还能完成大量“实际的”工作：捕获裸字、对有问题的构造生成警告、检查可能无法到达的代码、解析伪散列键，以及查找非法调用的子例程（即这些子例程的原型还未编译，就调用这些子例程）。

#### 第4趟：代码生成

这一趟是可选的。正常情况下并没有这一趟处理。不过，如果调用了3个代码生成器（`B::Bytecode`、`B::C`和`B::CC`）中的任意一个，则会最后访问一次解析树。代码生成器可能生成串行Perl字节码，用于以后重构解析树，也可能生成表示编译时解析树状态的原义C代码。

C代码的生成有两种不同方式。`B::C`只是重构解析树，并使用通常的`runops`循环来运行，Perl在执行过程中也在使用这个`runops`循环。`B::CC`生成运行时代码路径（构成一个庞大的跳转表）的一个线性的优化C版本，并执行这个代码。

在编译期间，Perl会采用很多种方法来优化代码。它会重新组织代码，使代码在执行时更高效。Perl会删除执行时永远也无法到达的代码，如`if (0)`块，或`if (1)`块中的`elsifs`和`else`。如果用采用`my ClassName$var`或`our ClassName $var`声明的词法作用域类型变量，而且`ClassName`包是用`use fields pragma`建立的，那么访问底层伪散列中的常量字段时，会在编译时完成拼写检查，并转换为数组访问。如果为`sort`操作符提供了一个足够简单的比较子例程，如`{ $a <=> $b }`或`{ $b cmp $a }`，这会替换为调用已编译的C代码。

Perl最显著的优化可能是它会尽可能早地解析常量表达式。例如，考虑图16-2所示的解析树。如果节点1和节点2都是直接量或常量函数，就会把节点1到节点4替换为计算结果，如图16-3所示。

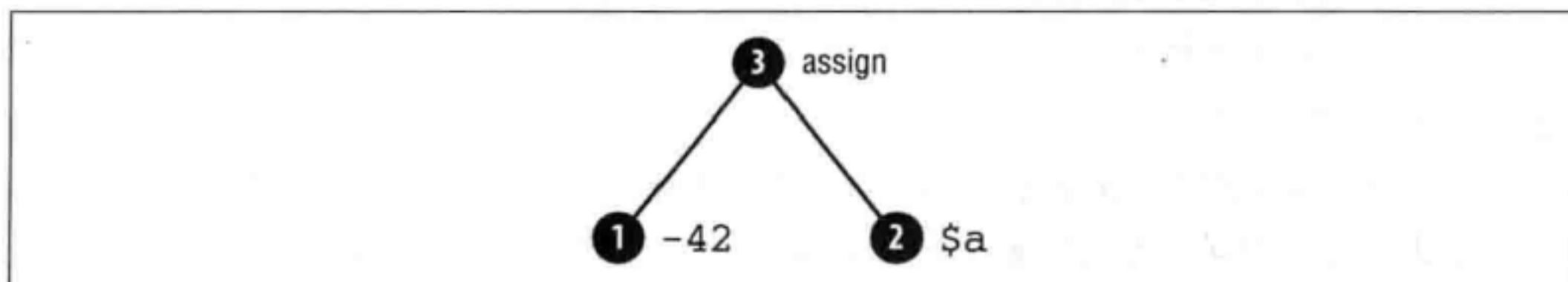


图16-3：常量叠算

这称为常量叠算（constant folding）。常量叠算并不仅限于这种情况（如在编译时把`2**10`转换为`1024`）。它还会解析函数调用，包括内置函数以及用户声明的子例程（要求满足第7章“内联常量函数”一节中介绍的原则）。FORTRAN编译器非常了解自己的内



部函数，算是对FORTRAN编译器的怀旧，Perl也知道编译中要调用哪个内置函数。正是因为这个原因，如果你想取0.0的对数，或者想计算一个负常量的平方根，会导致一个编译错误，而不是一个运行时错误，解释器根本不会运行<sup>注4</sup>。甚至任意复杂的表达式会提前解析，有时这会触发整个块的删除，如下所示：

```
if (2 * sin(1)/cos(1) < 3 && somefn()) { whatever() }
```

永远也不会计算的那些部分不会生成任何代码。因为第1部分总是false，所以somefn或whatever都不会被调用（所以不要指望用goto进入这个块中的标签，因为这个块在运行时甚至根本不存在）。如果somefn是一个可内联的常量函数，那么即使如下调换计算顺序：

```
if (somefn() && 2 * sin(1)/cos(1) < 3) { whatever() }
```

结果也没有任何改变，因为整个表达式仍在编译时解析。如果whatever是可内联的，运行时不会调用这个函数，甚至在编译时也不会，只是把它当作一个直接量常量插入它的值。然后你会得到一个警告，指出“Useless use of a constant in void context”（void上下文中使用常量没有意义）。如果你没有意识到它是一个常量，这可能会让你很奇怪。不过，如果whatever是非void上下文中调用（由优化器确定）的一个函数中最后计算的语句，你就不会看到这个警告。

用perl -Dx可以看到经过所有优化阶段之后所构造的解析树的最终结果（-D开关要求你使用一个特殊的支持调试的Perl构建版本）。参见“代码开发工具”一节中介绍的B::Deparse模块。

总之，Perl编译器会很努力地工作来优化代码（不过并不过分），使运行时整体执行速度提升。现在该运行你的程序了，下面就来讨论这个内容。

## 执行代码

大致说来，SPARC程序只在SPARC机器上运行，Intel程序只在Intel机器上运行，而Perl程序也只在Perl机器上运行。Perl机器拥有Perl程序所能找到的最理想的计算机特性：可以自动分配和释放的内存；没有大小限制的基本数据类型（如动态字符串、数组和散列等）；另外系统总有几乎同样的表现。Perl解释器的任务是：不论它在什么计算机上运行，都要让这台计算机看起来像是这种理想的Perl机器。

这个假想的机器给人一种错觉，就好像这是一个专门设计用来运行Perl程序的计算机。编译器生成的各个操作码是这个模拟指令集中的一个基本命令。并不是使用一个硬件程序计数器，解释器会跟踪记录要执行的当前操作码。这里没有使用一个硬件堆栈指针，解释器

---

注4： 其实我们这里有些过于简化。解释器实际上会运行，因为常量叠算器就是这样实现的。不过，它会在编译时立即运行，类似于BEGIN块的执行。



有自己的虚拟堆栈。这个堆栈非常重要，因为Perl虚拟机（我们不想把它称为PVM）是一个基于堆栈的机器。Perl操作码在内部被称为PP码（这是“压入-弹出（push-pop）码”的缩写），因为操作码要管理解释器的虚拟堆栈来找出所有操作数，处理临时值，并存储所有结果。

如果你曾用Forth或PostScript写过程序，或者用过采用RPN即逆波兰式（Reverse Polish Notation）的HP科学计算器，应该知道堆栈机是如何工作的。即使你没有接触过这些，也没有关系，这个概念很简单：要让3和4相加，实际的顺序是3 4 +，而不是我们更习惯的3 + 4。对于堆栈而言，这意味着你把3和4压入堆栈，然后+将这两个参数都弹出堆栈，让它们相加，再把结果7压回堆栈，它会一直留在栈中，直到你对它做其他的处理。

与Perl编译器相比，Perl解释器是一个非常直接、几乎让人厌烦的程序。它所做的就是逐个地处理已编译的操作码，并将它们分派到Perl运行时环境，也就是Perl虚拟机。它就是一堆枯燥的C代码，是吗？

实际上，它一点也不枯燥。Perl虚拟机会替你跟踪大量动态上下文，使你不用自己来跟踪。Perl维护了很多堆栈，你不必完全理解这些堆栈，不过下面还是列出这些堆栈，让你有点印象：

#### 操作数栈（operand stack）

这是我们已经讨论过的堆栈。

#### 保存栈（save stack）

这里保存局部值，以备以后恢复。很多内部例程也会在你不知道的情况下建立局部值。

#### 作用域栈（scope stack）

这是一个轻量级的动态上下文，它控制保存栈应当何时“弹出”。

#### 上下文栈（context stack）

这是一个重量级的动态上下文，谁调用了谁从而达到你当前所在的位置。caller函数遍历这个栈。循环控制函数扫描这个栈来查找要控制哪个循环。“回剥”上下文栈时，作用域栈也会相应回剥，这会从保存栈恢复所有局部变量，即使你用一些“邪恶”的方法离开之前的上下文（如产生一个异常，以及使用longjmp(3)）。

#### 环境跳转栈（jumpenv stack）

longjmp(3)上下文的堆栈，允许产生异常或迅速退出。

#### 返回栈（return stack）

记录进入这个子例程时我们来自哪里。

#### 标记栈（mark stack）

在操作数栈中当前可变参数列表从哪里开始。

递归词法填充栈 (*recursive lexical pad stacks*)

递归调用子例程时，词法作用域变量和其他“暂存寄存器”会保存在这里。

当然，还有一个存储所有C变量的C堆栈。实际上Perl会努力避免依赖C堆栈存储保存的值，因为`longjmp(3)`会绕过这些值的正常恢复。

介绍了这么多，我们所要说明的是，通常我们认为解释器作为一个解释另一个程序的程序，并不足以描述当前发生了什么。没错，确实有一些C代码来实现操作码，不过我们谈到“解释器”时，所说的并不只是这些，这就像我们谈到“音乐家”时，并不只是指可以把音符变成音乐的一组DNA指令。音乐家是“有状态”的活生生的有机生物。解释器也同样如此。

具体地，动态和词法上下文、全局符号表，再加上解析树，以及一个执行线程，所有这些才是我们所说的解释器。作为一个执行上下文，甚至在编译器开始之前解释器就已经存在了，甚至在编译器建立解释器的上下文时它就能以最基本形式运行。实际上，编译器调用解释器执行BEGIN块等代码时就是如此。解释器可以反过来使用编译器进一步构建。每次定义另一个子例程或加载另一个模块时，特定的Perl虚拟机（即我们所说的解释器）会重新定义自身。实际上不能说是编译器在控制，也不能说是由解释器来控制，因为它们会合作控制引导进程，即我们通常所说的“运行一个Perl脚本”。这就像启动一个孩子的大脑。这是由DNA做到的还是由神经细胞启动的？应该说二者都有一点，另外还需要外部程序员的一些输入。

可以在同一个进程中运行多个解释器。它们可能共享解析树，也可能不共享，这取决于它们是通过克隆一个现有的解释器来启动，还是从头开始构建一个新的解释器。也有可能在一个解释器中运行多个线程，在这种情况下，它们不仅共享解析树，还会共享全局符号。

不过，大多数Perl程序只使用一个Perl解释器来执行已编译的代码。尽管可以在一个进程中运行多个独立的Perl解释器，不过只能从C访问当前有关的API。各个Perl解释器相当于一个完全独立的进程，不过没有创建一个全新的进程开销那么大。Apache的`mod_perl`扩展性能如此突出正是出于这个原因：在`mod_perl`下启动一个CGI脚本时，这个脚本已经编译为Perl操作码，因此无需重新编译，不过更重要的是，这样就不必启动一个新的进程，这才是真正的瓶颈。Apache在一个现有的进程中初始化一个新的Perl解释器，并把之前编译的代码交给这个解释器来执行。当然，远不只是这些，它肯定还有更多工作。

还有很多其他应用（如`nvi`、`vim`和`innd`）可以嵌入Perl解释器。我们无法在这里一一列出。另外有很多商业产品甚至没有宣布它们嵌入了Perl引擎。这些产品是在内部使用Perl，因为Perl能漂亮地完成工作。



## 编译器后端

Apache可以先编译Perl程序，以后再执行这个程序，既然Apache能做到这一点，你为什么不能呢？Apache和包含内嵌Perl解释器的其他程序的做法很简单，它们从来都不把解析树存储到外部文件。如果你对这种做法很满意，而且不介意使用C API，也可以这样做。

如果你不想选择这条路，或者有其他需求，那么还有几个选择。不用将Perl编译器的操作码输出立即输入到一个Perl解释器，你可以调用某个后端模块。这些后端模块可以串行化已编译的操作码，并将其存储到一个外部文件，或者甚至可以把这些操作码转换为几种不同风格的C代码。

要当心代码生成器都还是处于试验阶段的工具，不要指望它们在生产环境中能正常工作。实际上，甚至在非生产环境中也不要期望它们能正常工作，除非有特殊情况。听我们这么讲，你的期望值应该已经很低了，现在任何成功都应该能让你心满意足，所以下面可以告诉你后端模块是如何工作的。

有些后端模块是代码生成器，如B::Bytecode、B::C和B::CC。另外一些实际上是代码分析和调试工具，如B::Deparse、B::Lint和B::Xref。除了这些后端模块，Perl标准版本还包括另外一些底层模块，如果你将来要编写Perl代码开发工具，可能会对这些模块感兴趣。另外在CPAN也能找到一些后端模块，（写这本书时）这些后端模块包括B::Fathom、B::Graph和B::Size。

如果使用Perl编译器的目的不是为解释器提供输入，而是做其他工作，编译器和各种后端模块之间会有一个O模块（即使用O.pm文件）。并不是直接调用这些后端模块；你要调用中间层，再由中间层调用指定的后端模块。所以，如果有一个名为B::Backend的模块，可以在一个给定的脚本中调用，如下所示：

```
% perl -MO=Backend SCRIPTNAME
```

有些后端模块还可以指定选项：

```
% perl -MO=Backend,OPTS SCRIPTNAME
```

有些后端模块已经有自己的前端来为你调用其中间层，所以你不用记住它们的M.O。尤其是perlcc(1)会调用这个代码生成器（启动它可能比较麻烦）。

## 代码生成器

当前有3种后端模块可以将Perl操作码转换为其他某种格式，这3个后端模块都还处于试验阶段（没错，之前我们提到过这一点，不过这里重申一次，以免你忘记）。即使生成的输出恰好能正确地运行，所得到的程序也有可能存在一些问题，如与正常情况相比，它可能



占用更多磁盘空间、更多内存以及更多CPU时间。这还是一个正在研究和开发的领域。情况会越来越好的。

## 字节码生成器

`B::Bytecode`模块将解析树的操作码用一种平台无关的编码格式写出。可以把Perl脚本编译为字节码，然后把它复制到另外一台安装有Perl的机器上。

标准的`perlcc(1)`命令（不过当前还处于试验阶段）知道如何将Perl源代码转换为字节码Perl程序。你要做的就是调用以下命令：

```
% perlcc -B -o pbyscript srcscript
```

现在应该能直接“执行”所得到的pbyscript。这个文件的开头可能如下所示：

```
#!/usr/bin/perl
use ByteLoader 0.03;
^C^@E^A^C^@^@^A^F^@C^@^@B^F^@C^@^@C^F^@C^@^@
B^@^@^H9^A8M-^?M-^?M-^?M-^?M-^?M-^?M-^?6^@^@A6^@
^G^D^D^@^@^KR^@^@^HS^@^@^HV^@M-2<W^FU^@^@^X^Y@Z^@
...
```

你会发现一个小的脚本首部，后面是纯二进制数据。这看起来很神秘，不过实际上这只是一个很小的“法术”。ByteLoader模块使用了一种称为“源码过滤器”（source filter）的技术，在Perl看到源代码之前先对源码进行修改。源码过滤器是一种预处理器，将应用于当前文件中在它后面出现的所有代码。类似`cpp(1)`和`m4(1)`的宏处理器只会做一些简单的转换，而源码过滤器没有任何限制。源码过滤器可以用来检查Perl的语法，压缩或加密源代码，甚至可以用Latin. E perlibus unicode等奇怪的格式转换源代码。不过，写脚本者自己要慎重。

ByteLoader模块是一个源码过滤器，知道如何解开`B::Bytecode`生成的串行操作码来重新构造原来的解析树。重构的Perl代码可以接入到当前解析树而无需使用编译器。解释器看到这些操作码时就会执行，就好像这些操作码一直在解析树中一样。

## C代码生成器

另外两个代码生成器`B::C`和`B::CC`都生成C代码而不是串行Perl操作码。它们生成的代码很难读，如果你想读这些代码，会完全不明所以。不要以为这只是把从Perl转换到C的一些代码片段插入到一个更大的C程序，使用起来可没有那么简单。

`B::C`模块会写出重建整个Perl运行时环境所需的C数据结构。你会得到一个专用的解释器，编译器构建的所有数据结构已经预初始化。从某种意义上讲，所生成的代码与`B::Bytecode`生成的代码有些类似。它们都是对编译器构建的操作码树进行直接转换，不

过B::Bytecode会采用一种符号形式，以便以后重建并插入到一个运行的Perl解释器，B::C则采用C表示这些操作码。用C编译器编译这个C代码并链入Perl库时，得到的程序不再要求目标系统上安装有一个Perl解释器（不过，如果没有完全采用静态链接，可能还需要一些共享库）。但这个程序与运行脚本的常规Perl解释器并没有根本上的区别。它只是预编译为一个独立的可执行映像。

不过，B::CC模块要做的工作更多。它生成的C源文件的开头看起来与B::C生成的相应部分很类似<sup>注5</sup>，但它们相似的地方仅此而已。在B::C代码中，会有一个很大的C操作码表，要像解释器那样处理，而B::CC生成的C代码的顺序与程序运行时流的顺序对应。对于程序中的各个函数甚至分别有一个相应的C函数。B::CC还会根据变量类型完成一定的优化；与标准解释器相比，一些基准测试的运行速度可以快一倍。这是当前代码生成器中最强大的一个，它对未来做出了最大的承诺。并不是巧合，它也是这三个代码生成器中最不稳定的一个。

计算机科学专业的学生完全可以从这里寻找毕业研究课题，这里有很多尚未雕琢的宝石等着你们来挖掘。

## 代码开发工具

除了为试验性的代码生成器提供输入，O模块还有很多有意思的应用。由于允许轻松地访问Perl编译器的输出，利用这个模块可以很容易地构建其他需要完全了解Perl程序的工具。

B::Lint模块由*lint*(1)得名，这是一个C程序校验器。它会检查程序中有问题的构造，这些构造会让初学者很困惑，但通常并不触发警告。可以直接调用这个模块：

```
% perl -MO=Lint,all myprog
```

目前只定义了为数不多的一些检查，如在一个隐式标量上下文中使用数组，依赖于默认变量，以及访问另一个包中以\_开头的标识符（通常是私有标识符）。有关详细内容参见B::Lint(3)。你可能会发现，大多数Perl程序员写程序时都会使用Perl::Critic。这是一个在PPI基础上构建的静态分析工具，它是一个很棒的工具。

对于程序中的所有变量（包括全局变量和词法作用域变量）、子例程和格式，B::Xref模块可以生成其声明和使用的交叉引用列表，按文件和子例程划分。可以如下调用这个模块：

```
% perl -MO=Xref myprog > myprog.pxref
```

---

注5： 不过，如果你完全看不懂，那么所有代码看起来都一样。我们警告过你吧？不要看这些代码！

例如，下面是报告的部分输出：

```
Subroutine parse_argv
  Package (lexical)
    $on i113, 114
    $opt i113, 114
    %getopt_cfg i107, 113
    @cfg_args i112, 114, 116, 116
  Package Getopt::Long
    $ignorecase 101
    &GetOptions &124
  Package main
    $Options 123, 124, 141, 150, 165, 169
    %$Options 141, 150, 165, 169
    &check_read &167
    @ARGV 121, 157, 157, 162, 166, 166
```

这说明，`parse_argv`子例程有自己的4个词法作用域变量，还访问了来自`main`包和`Getopt::Long`的一些全局标识符。这里的数字是使用这些项的行号：前导`i`表示这一项在下一个行号首次引入，前导`&`表示这里调用了一个子例程。解引用分别列出，所以这里`$Options`和`%$Options`都会显示。

`B::Deparse`是一个很不错的“打印机”，可以消除Perl代码的神秘感，帮助你理解优化器对代码做了哪些转换。例如，下面显示了Perl对各种构造使用的默认设置：

```
% perl -MO=Deparse -ne 'for (1 .. 10) { print if -t }'
LINE: while (defined($_ = <ARGV>)) {
    foreach $_ (1 .. 10) {
        print $_ if -t STDIN;
    }
}
```

`-p`开关会增加小括号，使你能看到Perl的优先级设置：

```
% perl -MO=Deparse,-p -e 'print $a ** 3 + sqrt(2) / 10 ** -2 ** $c'
print(((( $a ** 3) + (1.4142135623731 / (10 ** -(2 ** $c))))));
```

可以使用`-q`来了解基本内插字符串将编译得到什么结果：

```
% perl -MO=Deparse,-q -e '"A $name and some @ARGV\n"'
'A ' . $name . ' and some ' . join("$", @ARGV) . "\n";
```

下面给出了Perl如何将一个三部分for循环编译为一个while循环：

```
% perl -MO=Deparse -e 'for ($i=0;$i<10;$i++) { $x++ }'
$i = 0;
while ($i < 10) {
    ++$x;
}
continue {
    ++$i
}
```



甚至可以对`perlcc -b`生成的Perl字节码文件调用`B::Deparse`，让它为你反编译这个二进制文件。串行化Perl操作码可能很难读，不过它们并不是高强度的加密。

## 先编译，后解释

总要有一个时间全面考虑所有问题，有时要前期考虑，有时需要后期考虑。有些情况下，还可能要在中间考虑。Perl没有假定什么时候考虑最合适，所以它为程序员提供了很多选项，允许程序员告诉它什么时候考虑这些问题。还有一些情况下，它知道必须做一些考虑，但是对于应当考虑哪些问题毫无想法，所以它需要一些方法来询问你的程序。你的程序要回答这些问题，为此需要定义有指定名字的子例程，这些子例程名要与Perl想要查找的答案对应。

编译器想要提前考虑时可以调用解释器，不仅如此，解释器想要修改历史时也可以反过来调用编译器。你的程序可以利用很多操作符回过头来调用编译器。类似于编译器，解释器想要查找某些问题时也可以调用命名子例程。由于编译器、解释器以及你的程序之间存在这些“来回交互”。你要知道什么时候发生了什么。首先，我们会讨论什么时候触发这些命名子例程。

第10章中我们已经看到，在一个包中调用一个未定义函数时就会触发这个包的AUTOLOAD子例程。在第12章中我们见过DESTROY方法，对象的内存将由Perl自动回收时就会调用这个方法。另外在第14章中，我们已经见过访问一个绑定变量时会隐式调用很多函数。

这些子例程都遵循一个约定，即如果一个子例程由编译器或解释器自动触发，这个子例程名就用大写。另外还有几个子例程与程序生命期的不同阶段相关，分别名为BEGIN、UNITCHECK、CHECK、INIT和END。这些子例程声明前面的sub关键字是可选的。可能它们叫做“模块”更合适，因为在某些方面它们更像是命名模块而不是真正的子例程。

例如，与常规子例程不同，多次声明这些代码块没有任何问题，因为Perl会跟踪何时调用这些代码块，这样你就不必通过名字来调用（它们在另一个方面也与常规子例程不同，`shift`和`pop`表现得就像在主程序中一样，所以它们默认地作用于`@ARGV`而不是`@_`）。

这5类代码块按以下顺序运行：

### BEGIN

一旦解析就运行，即只要在编译时遇到，则在编译文件的其余部分之前立即运行。

### UNITCHECK

定义这些模块的单元完成编译后就运行。主程序文件和它加载的各个模块都是编译单元，字符串`eval`、正则表达式中使用`(?{ })`和`(??{ })`构造编译的代码、`do FILE`和`require FILE`调用，以及命令行上`-e`开关后面的代码也都是编译单元。你可能更希望使用这一类模块来运行初始化代码而不是INIT。

## CHECK

在编译完成之后但在程序开始之前运行（CHECK可能表示“检查点”或“双重检查”，或者甚至表示“停止”）。

## INIT

在程序主流程即将开始之前运行。

## END

在程序完成之后运行。

如果将以上代码块声明多次（同名），即使是在不同的模块中，BEGIN都会在CHECK之前运行，CHECK都在INIT之前运行，INIT都在END之前运行，当然END都在主程序完成之后最后运行。多个BEGIN和INIT会按声明顺序（FIFO）运行，CHECK和END则按声明的相反顺序（LIFO）运行。

通过一个演示示例可以很容易看到具体的运行顺序：

```
use v5.10;
say      "start main running here";
die      "main now dying here\n";
die      "XXX: not reached\n";
UNITCHECK { say "1st UNITCHECK: done compiling" }
END      { say "1st END: done running" }
CHECK    { say "1st CHECK: done compiling" }
INIT     { say "1st INIT: started running" }
END      { say "2nd END: done running" }
BEGIN    { say "1st BEGIN: still compiling" }
INIT     { say "2nd INIT: started running" }
BEGIN    { say "2nd BEGIN: still compiling" }
CHECK    { say "2nd CHECK: done compiling" }
END      { say "3rd END: done running" }
```

运行时，这个演示程序会生成以下输出：

```
1st BEGIN: still compiling
2nd BEGIN: still compiling
1st UNITCHECK: done compiling
2nd CHECK: done compiling
1st CHECK: done compiling
1st INIT: started running
2nd INIT: started running
start main running here
main now dying here
3rd END: done running
2nd END: done running
1st END: done running
```

因为BEGIN块会立即执行，它可能在编译文件的其余部分之前引入了其他子例程声明、定义和导入。这些可能影响编译器对当前文件其余部分的解析，特别是如果你导入了子例程定义的情况下。至少，声明一个子例程就允许将它用作为一个列表操作符，这使得小括号



变得可有可无。如果导入的子例程声明有一个原型，对这个子例程的调用可能会像内置函数一样解析，甚至可以覆盖同名的内置函数从而为它们提供不同的语义。`use`声明就是一个会带来影响的BEGIN块。

与之相反，END块会尽可能晚地执行：当程序退出Perl解释器时才会执行，甚至可能由于一个未捕获的die或其他致命异常触发END块执行。有两种情况会跳过END块（或DESTROY方法）。如果程序并不是退出，而是当前进程通过exec从一个程序变换到另一个程序，此时END不会运行。如果一个进程由于未捕获的信号而被终止，也会跳过其END例程（参见第29章介绍的sigtrap pragma，其中提供了一种简单的方法可以将可捕获的信号转换为异常。关于信号处理的一般信息，请参见第15章的“信号”一节）。要想避免所有END处理，可以调用POSIX::\_exit，如kill -9, \$\$，或者使用exec执行某个“无关痛痒”的程序，如Unix系统上的/bin/true。

在一个END块中，\$?包含程序退出时的状态。可以在END块中修改\$?来改变程序的退出值。要当心由于使用system或反引号运行另一个程序而无意地改变了\$?。

如果一个文件中有多个END块，它们会按其定义的相反顺序执行。也就是说，最后一个定义的END块将在程序完成时第一个执行。通过逆转顺序，这就允许相关的BEGIN和END块（如果正确配对）可以按你期望的方式嵌套。例如，如果主程序和它加载的一个模块都有自己的成对使用的BEGIN和END子例程，如下所示：

```
BEGIN { print "main begun" }
END { print "main ended" }
use Module;
```

在加载的这个模块中，有以下声明：

```
BEGIN { print "module begun" }
END { print "module ended" }
```

主程序会知道它的BEGIN总是最先发生，它的END总是最后发生（没错，BEGIN实际上是一个编译时代码块，不过在运行时，配对的INIT和END块也同样有这个特性）。对于任何包含另一个文件的文件，如果二者都有类似这样的声明，那么这个原则总成立。这种嵌套特性使这些代码块就类似于包构造方法和析构方法。每个模块可以有自己的建立和撤销函数，Perl将自动调用这些函数。这样一来，程序员不必记住是否使用某个特定的库、应当调用哪个特殊的初始化或清理代码，或者何时调用这些代码。这些事件可以由模块的声明来保证。

如果把一个eval STRING看作是从解释器到编译器的反向调用，那么可以认为BEGIN是从编译器到解释器的一个正向调用。它们都会临时将当前活动置于挂起状态，并切换操作模式。我们说一个BEGIN块尽可能早地执行，这是指一旦它完全定义就立即执行，甚至要在解析当前文件的其余部分之前执行。因此BEGIN块会在编译时执行，而不是在运行时执行。一旦运行了一个BEGIN块，它会立即取消定义，它使用的所有代码则返回到Perl的内存池。不能将BEGIN块作为一个子例程来调用，因为在你调用时，它已经不存在了。



与BEGIN块类似，INIT块会在Perl运行时系统开始以“先进先出”（FIFO）顺序执行之前运行。例如，*perlcc*中描述的代码生成器就利用INIT块来初始化并解析XSUB指针。INIT块实际上类似于BEGIN块，只不过允许程序员区别对两类构造加以区分，一类是必须在编译阶段发生，另一类必须在运行阶段发生。直接运行一个脚本时，这个区别并不特别重要，因为每次都会调用编译器。不过如果编译与执行分开，这个区别就很重要了。编译器可能只调用一次，而得到的可执行程序可能会调用多次。

与END块类似，CHECK块在Perl编译阶段结束之后而且运行阶段开始之前以LIFO顺序运行。CHECK块对于“结束”编译器很有用，这类似于END块对于“结束”程序的作用。具体来讲，后端模块都使用CHECK块作为钩子，从它调用各自的代码生成器。它们只需要把一个CHECK块放在自己的模块中，它就会在适当的时候运行，所以你不必在程序中安装一个CHECK。出于这个原因，你很少自己写CHECK块，除非需要写这样一个模块。

汇总以上内容，表16-2列出了各个构造，并提供了它们何时编译以及何时运行代码等详细信息（代码用“...”表示）。

表16-2：不同时刻发生的不同活动

块或表达式	哪个阶段	跟踪编译	哪个阶段	跟踪运行	调用触发
	编译	错误	运行	错误	器策略
use ...	编译	否	编译	否	现在
no ...	编译	否	编译	否	现在
BEGIN {...}	编译	否	编译	否	现在
UNITCHECK {...}	编译	否	编译	否	晚
CHECK {...}	编译	否	编译	否	晚
INIT {...}	编译	否	运行	否	早
END {...}	编译	否	运行	否	晚
eval {...}	编译	否	运行	是	内联
eval ..." "	运行	是	运行	是	内联
foo(...)	编译	否	运行	否	内联
sub foo {...}	编译	否	运行	否	任何时间调用
eval sub "{...} "	运行	是	运行	否	以后调用
s/pat/.../e	编译	否	运行	否	内联
s/pat/..."/ee "	运行	是	运行	是	内联

既然你已经知道了编译是怎么回事，希望你能更有信心地编写和完成你的Perl程序。

# 命令行接口

这一章的目的是教你在真正开火之前先让Perl瞄准方向。让Perl瞄准方向的方法有很多，不过有两种主要的方法：通过命令行开关以及通过环境变量。开关是瞄准某个特定命令最直接、最精确的方法。环境变量更常用于设置一般策略。

## 命令处理

Perl是在Unix世界成长起来的，这真是很幸运，因为这意味着它的调用语法在其他操作系统的命令解释器中也能用。大多数命令解释器都知道如何将一系列单词处理为参数，而且不介意参数以负号开头。当然，从一个系统转向另一个系统时也会遇到一些棘手的问题。例如，在MS-DOS系统上不能像在Unix上那样使用单引号。另外，在诸如VMS等系统上，一些包装器代码必须经过处理才能模拟Unix I/O重定向。通配符的解释也是形形色色，千差万别。不过，一旦解决了这些问题，Perl在任何操作系统上都能以同样的方式处理开关和参数。

即使你没有命令解释器，也能很容易地从用任何语言编写的另一个程序执行Perl程序。调用程序不仅可以采用正常的方式传递参数，也可以通过环境变量传递信息，如果你的操作系统提供相应支持，甚至还可以通过继承的文件描述符来传递信息（参见第15章“传递文件句柄”一节）。甚至可以很容易地将一些外来的参数传递机制封装在模块中，然后通过一个简单的use指令引入你的Perl程序。

Perl采用标准的流行方式<sup>注1</sup>解析命令行开关。也就是说，它希望命令行上首先出现的是开关（以负号开头）。开关后面通常是脚本名，接下来是传入这个脚本的任何其他参数。其中一些参数本身看起来可能很像开关，不过即使是这样，它们也必须由脚本来处理，因为

注1： 假设你认同Unix既标准又流行。

Perl一旦看到一个非开关或者特殊的“--”开关（它表示“我是最后一个开关”），Perl就会停止解析开关。

关于程序源代码放在哪里，Perl提供了一定的灵活性。对于短小、快捷的任务，可以直接在命令行编写Perl程序。对于比较大、更长久的任务，可以提供一个Perl脚本作为单独的文件。Perl将用以下三种方式查找要编译和运行的脚本：

1. 在命令行上通过-e或-E开关逐行指定。例如：

```
% perl -e "print 'Hello, World.'"
Hello, World.
```

```
% perl -E "say 'Howdy y\\'all!'"
Howdy y'all!
```

2. 包含在命令行上第一个文件名指定的文件中。如果系统支持可执行脚本的第一行使用#!记法，这些系统就可以为你以这种方式调用解释器。
3. 通过标准输入隐式传入。这个方法只适用于没有提供文件名参数的情况；要向一个标准输入脚本传递参数，必须使用方法2，并显式地指定一个“-”作为脚本名。  
例如：

```
% echo "print qq(Hello, @ARGV.)" | perl - World
Hello, World.
```

对于方法2和方法3，Perl会从头开始解析输入文件，除非你指定了一个-x开关，在这种情况下，它会扫描第一行以#!开头并包含单词“perl”的代码，然后从那里开始解析。如果要运行的脚本嵌入在一个更大的消息中，这个开关会很有用。如果是这样，还可以使用\_\_END\_\_ token指示脚本末尾。

不论是否使用-x，解析行时总会检查#!行是否包含开关。这样一来，如果你所在的平台只允许#!行有一个参数，或者更糟糕，甚至不能把#!行识别为一个特殊的行，你仍能得到一致的开关行为，而不论Perl以何种方式调用（即使使用-x来查找脚本的开头）。

警告：由于老版本的Unix会不声不响地“砍掉”#!行内核解释超过32字符的部分，有些开关可能会完整地传入程序，有些开关则不能；如果不当心，甚至可能会传入一个“-”而没有相应的字母。你可能想确保所有开关都落在这个32字符边界前面，或者都在这个边界后面。大多数开关并不关心是否冗余地处理，不过如果得到一个“-”而不是一个完整的开关，可能导致Perl尝试从标准输入读取其源代码，而不是从你的脚本读取。另外，不完整的-l开关还可能导致奇怪的结果。不过，有些开关确实关心是否被处理两次，如-l和-O组合。要么把所有开关都放在32字符边界的后面（如果可行），要么不使用-oDIGITS而替换为BEGIN{ \$/ = "\oDIGITS"; }。当然，如果不在Unix系统上，你肯定不会遇到这个特殊问题。



对`#!`开关的解析是从行中首次提到“perl”的位置开始的。会对`emacs`用户特别忽略序列“`-*`”和“`-`”，因此如果你要使用`emacs`，可以写为：

```
#!/bin/sh -- # -*- perl -*- -p
eval 'exec perl -S $0 ${1+"$@"}'
if 0;
```

Perl只会看到`-p`开关。“`-*- perl -*-`”告诉`emacs`以Perl模式启动；如果不使用`emacs`，就不需要这一部分。`-S`将在后面介绍这个开关时再做解释。

`env(1)`程序（如果你有这个程序）可以使用类似的技巧：

```
#!/usr/bin/env perl
```

前面的例子会使用Perl解释器的相对路径，这会得到用户路径中第一个出现的版本。如果想要某个特定的Perl版本，比如`perl5.14.0`，可以把它直接放在`#!`行的路径中，为此可以利用`env`程序，或者使用`-S`，也可以利用常规的`#!`处理。

如果`#!`行不包含单词“perl”，将执行`#!`后指定的程序而不是Perl解释器。例如，假设你有一个普通的Bourne shell脚本，如下所示：

```
#!/bin/sh
echo "I am a shell script"
```

如果把这个文件交给Perl，Perl会为你运行`/bin/sh`。这有点奇怪，不过如果有人使用的机器不能识别`#!`，这对他们就很有帮助，因为通过设置其SHELL环境变量，可以告诉程序（如邮件程序）他们的shell是`/usr/bin/perl`。然后Perl可以为他们将这个程序分派到正确的解释器（尽管他们自己无法做到）。

不过再来看真正的Perl脚本。找到你的脚本之后，Perl把整个程序编译为一种内部形式（参见第16章）。如果出现任何编译错误，执行甚至根本不会开始（这与一般的shell脚本或命令文件不同，发现语法错误时有可能已经运行了一半）。如果脚本的语法是正确的，就会执行这个脚本。如果脚本正常运行直到最后结束，而没有遇到任何`exit`或`die`操作符，则由Perl提供一个隐式的`exit(0)`，指示调用者脚本成功执行（这与典型的C程序也不同，在C程序中，如果程序正常终止，你可能会得到一个随机的退出状态）。

## 非Unix系统上的`#!`和引号

可以在其他系统上模拟Unix的`#!`技术：

### MS-DOS

创建一个批文件来运行程序（用`ALTERNATIVE_SHEBANG`方式编写）。有关的更多信息，请参见Perl源代码发布版顶级目录中的`dosish.h`文件。

## OS/2

将下面这行代码：

```
extproc perl -S -your_switches
```

作为第一行放在\*.cmd文件中（-S可以绕过cmd.exe的“extproc”处理中的一个bug）。

## VMS

将下面这两行代码：

```
% perl -mysw 'f$env("procedure")' 'p1' 'p2' 'p3' 'p4' 'p5' 'p6' 'p7' 'p8' !  
$exit++ + ++$status != 0 and $exit = $status = undef;
```

放在程序最前面，这里-mysw是你想传入Perl的任意命令行开关。现在可以输入perl program直接调用这个程序，或者通过@program将程序作为一个DCL过程来调用，或者只使用程序名通过DCL\$PATH隐式调用。这就像“咒语”，有点不太好记，不过如果在perl中键入“-V:start perl”，Perl会为你显示这个咒语。如果你连这个也记不住，没关系，这正是你买这本书的原因。

## Windows

使用Perl的ActiveState版本时，如果在Microsoft的Windows系列操作系统上（也就是Win95、Win98、Win00<sup>注2</sup>、WinNT，但不包括Win3.1），Perl的安装过程会修改Windows注册表，将.pl扩展名与Perl解释器关联。

如果你安装了Perl的另一个移植版本，包括Perl的Win32目录中的那个移植版本，就必须自行修改Windows注册表。

需要注意，使用.pl扩展名意味着你将无法区分可执行Perl程序和“perl库”文件。可以对Perl程序使用.plx来避免这个问题。如今这已经不算是太大的问题，因为Perl模块都放在.pm文件中，人们不会再写太多的.pl文件。

对于如何加引号，非Unix系统上的命令解释器与Unix shell的做法通常有很大不同。你需要了解你的命令解释器有哪些特定字符（比较常见的有\*、\和"），另外要了解如何保护空白符和这些特殊字符通过-e开关来运行单行脚本。还可能要把单个的%改为%%，否则就会对它转义（如果这对你的shell来说是一个特殊字符）。

在一些系统上，可能要把单引号改为双引号。不过在Unix或Plan9系统上不要这么做，或者运行Unix风格的shell的任何系统上都不要这么做（如来自MKS Toolkit的系统，或者来自Cygwin包的系统，Cygwin原先由Cygnus出品，现在收归Redhat所有）。Microsoft提供了一个名为Interix的新Unix模拟器，也开始注意到这个问题。

例如，在Unix（包括Linux和Mac OS X）上，可以使用：

---

注2： 嗯，请原谅由于技术难度，所以……

```
% perl -e 'print "Hello world\n"'
```

在VMS上要使用：

```
$ perl -e "print ""Hello world\n"""
```

或者也可以利用qq//：

```
$ perl -e "print qq(Hello world\n)"
```

另外在MS-DOS等系统上，可以使用：

```
A: perl -e "print \"Hello world\n\""
```

或者使用qq//来选择你自己的引号：

```
A: perl -e "print qq(Hello world\n)"
```

问题在于，这些方法都是不可靠的：这取决于你使用的命令解释器。并没有一个适用所有系统的通用解决方案，现在还只是一团糟。如果你使用的是一个Unix系统，但是想完成一些命令行工作，最好去找一个更好的命令解释器，而不是直接使用你的供应商提供的命令解释器，这应该不会太难。

或者，也可以都用Perl来编写，完全忘掉这些单行脚本。

## Perl的位置

尽管看起来可能很明显，不过还是需要指出：只有当用户能够很容易地找到Perl时，Perl才真正有用。如果可能，最好`/usr/bin/perl`和`/usr/local/bin/perl`都是指向实际二进制文件的符号链接。如果做不到，强烈建议系统管理员把Perl及其相关工具放在通常可以在用户标准PATH中找到的某个目录中，或者放在另外某个明显而方便的位置。

在本书中，我们在程序的第一行使用标准`#!/usr/bin/perl`记法来表示你的系统上所采用的任何特定机制。如果你想要运行某个特定版本的Perl，可以使用一个特定的路径：

```
#!/usr/local/bin/perl5.14.0
```

如果你只是想运行至少某个版本的Perl，而不关心版本是否更高，可以在程序开始位置附近放置类似下面的语句：

```
use v5.14;
```

注意：Perl很早前的版本使用类似“5.005”或“5.004\_05”的版本号。如今我们可以把它们看作是v5.5.0和v5.4.5，不过比v5.6.0早的Perl版本不理解这种记法。`use 5.NNN`形式是最安全的，可以确保向后兼容性，甚至可以与千年前的版本兼容。



## 开关

单字符命令行开关如果没有自己的参数，往往会与它后面的一个开关组合（捆绑）在一起。

```
#!/usr/bin/perl -spi.bak # 等同于-s -p -i.bak
```

开关也称为选项或标志。不论它们叫什么，这并不重要。下面给出Perl能识别的一些开关：

-- 终止开关处理，即使下一个参数以一个负号开头。这个开关没有其他作用。

### -ODIGITS

将输入记录分隔符（\$/）指定为表示该字符码点的一个八进制数字或十六进制数字。如果没有指定八进制或十六进制数，则null字符（即U+0000，Perl的"\0"）为分隔符。在这个八进制或十六进制数前面或后面可能还有其他开关。例如，如果有一个find(1)可以打印以null字符结束的文件名，那么可以用以下命令来删除：

```
% find . -name '*.bak' -print0 | perl -n0e unlink
```

指定值00使Perl以段落模式读取文件，等价于将\$/变量设置为""。0400或大于0777的任何值都会使Perl一次“吞入”整个文件，不过按约定，这里通常使用值0777。这等价于取消\$/变量的定义。我们使用0777的原因是没有与之等值的ASCII字符（遗憾的是，确实有一个与之等值的Unicode字符，即LATIN SMALL LETTER O WITH STROKE AND ACUTE，不过有人告诉我们不能用这个字符来分隔记录。但如果你确实想要这么做，可以用十六进制指定它的码点：-0x1FF）。

还可以使用十六进制记法指定分隔符：-0xHHH...，这里“H”是合法的十六进制数。与八进制形式不同，这种记法可以用来指定任何Unicode字符，甚至超过0xFF的字符（这说明，不能对仅包含十六进制数的目录名使用-x开关）。

-a 打开自动分解模式，不过只能结合-n或-p使用。在-n和-p开关生成的隐式while循环中，首先对@F数组执行一个隐式split命令。所以：

```
% perl -ane 'print pop(@F), "\n";'
```

等价于：

```
LINE: while (<>) {  
    @F = split(' ');  
    print pop(@F), "\n";  
}
```

可以为-F开关传入要分解的正则表达式，来指定一个不同的字段分隔符。例如，下面这两个调用是等价的：

```
% awk -F: '$7 && $7 !~ /^\/bin/' /etc/passwd  
% perl -F: -lane 'print if $F[6] && $F[6] !~ m(^/bin)' /etc/passwd
```

- c 要求Perl检查脚本的语法，然后退出，而不执行它刚才编译的程序。从技术角度讲，它做的工作并不止这些，它还会执行所有BEGIN、UNITCHECK和CHECK块，以及所有use或no指令，因为这些都可以认为在程序执行之前发生。不过，它不执行任何INIT或END块。要想得到以前那种用处不大的行为，只需把

```
BEGIN { $^C = 0; exit; }
```

放在主脚本的末尾。这是因为\$^C变量反映了-c开关的值。

## -C [number/list]

-C标志会控制Perl的某些Unicode特性。-C后面可以是一个数字，也可以是一个选项字母列表。字母、相应数值及其影响如表17-1所示，这里罗列字母等价于数字求和。

表17-1：-C开关的值

字母	十六进制数	含义
<i>I</i>	0x1	假设STDIN采用UTF-8格式
<i>O</i>	0x2	STDOUT将采用UTF-8格式
<i>E</i>	0x4	STDERR将采用UTF-8格式
<i>S</i>	0x7	I + O + E
<i>i</i>	0x8	UTF-8是输入流的默认PerlIO层
<i>o</i>	0x10	UTF-8是输出流的默认PerlIO层
<i>D</i>	0x18	i + o
<i>A</i>	0x20	@ARGV元素是用UTF-8编码的字符串
<i>L</i>	0x40	通常“IOEioA”是无条件的；L使它们成为有条件的，依赖于本地化环境变量（LC_ALL、LC_TYP和LANG，这里按优先级递减的顺序列出）。如果变量指示UTF-8，则选择的“IOEioA”生效
<i>a</i>	0x100	设置\${^UTF8CACHE}为-1将以调试模式运行UTF-8缓存代码。这可能没有任何意义，除非你想调试或重写Perl的内部模块

例如，-COE及其等价的数值表示-C6将在STDOUT和STDERR上启用UTF-8格式。重复的字母只是冗余，而没有累加作用，也不表示切换。

io选项表示当前文件作用域中所有后续的open（或类似的I/O操作）都会导致对其隐式地应用:utf8 PerlIO层；换句话说，输入流会采用UTF-8，输出流也会使用UTF-8。这只是默认情况，如果在open中显式地设置层或者使用binmode，也可以像平常一样管理流。

如果只有-C本身（后面没有跟其他数字或选项列表），或者PERL\_UNICODE环境变量

使用空串""，这与-CSDL有同样的效果。换句话说，标准I/O句柄和默认的open层都使用utf8，不过前提是与本地化环境相关的环境变量指示一个UTF-8本地化环境。这种行为遵循v5.8.0的隐式UTF-8行为，现在不建议使用。

可以使用-C0（或PERL\_UNICODE环境变量为"0"）来显式地禁用以上Unicode特性。

魔法变量\${^UNICODE}反映了这个设置的数字值。这个变量在Perl启动时设置，而且在此之后是只读的。如果你想要运行时设置，可以使用open pragma，或者使用open的三参数形式或binmode的两参数形式。

在v5.8.1之前的版本中，-C开关是一个仅Win32支持的开关，用来启用能够处理Unicode的“全系统调用”Win32 API。这个特性在实际中几乎不用，所以这个命令行开关被“回收”另做他用。

注意：从v5.10.1版本开始，如果#!行上使用了-C选项，也会在命令行上指定。这是因为Perl解释器执行时已经建立了标准流。还可以使用binmode设置I/O流的编码。

-d

-dt 在Perl调试器中运行程序。参见第18章。如果指定了t，则指示调试器：所调试的代码中将使用线程。

-d: MODULE[=ARG1,ARG2]

-dt: MODULE[=ARG1,ARG2]

在一个调试、性能测试或跟踪模块（安装为Devel::MODULE）控制之下运行程序。例如，-d:DProf使用Devel::DProf性能测试模块执行程序。类似于-M标志，可以将选项传入Devel::MODULE包，在这里将由Devel::MODULE::import例程接收和解释这些选项。同样的，与-M类似，可以使用-d:-MODULE调用Devel::MODULE::unimport而不是import。逗号分隔的选项列表必须放在一个=字符后面。如果指定了t，则指示调试器：所调试的代码中将使用线程。

-D LETTERS

-D NUMBERS

设置调试标志（只有如下所述在你的Perl版本中编译了调试特性时这才有效）。可以指定一个NUMBER（你想要的位的总和），或者可以指定一个LETTERS列表。例如，要查看如何执行你的脚本，可以使用-D14或-Dslt。另一个有用的值是-D1024或-Dx，这会列出已编译的语法树。-D512或-Dr会显示已编译的正则表达式。数字值可以通过特殊变量\$^D在内部得到。表17-2列出了指定的位值。下面的数字按十六进制给出，以便阅读，不过如果你使用-NUMBER格式，必须按十进制来提供。强烈推荐使用字母。



表17-2: -D选项

位	字母	含义
0x0400000	<i>A</i>	对内部结构的一致性检查 (“警报全清除了吗?”)
0x2000000	<i>B</i>	转储子例程定义, 包括类似BEGIN的特殊块
0x0200000	<i>C</i>	写时复制
0x0000020	<i>c</i>	字符串-数字转换
0x0008000	<i>D</i>	一旦程序完成则清理
0x0000100	<i>f</i>	格式处理
0x0002000	<i>H</i>	散列转储: usurps values
0x0080000	<i>J</i>	对包DB::中的操作码显示s,t,P-debug (也就是说, 不跳过)
0x0000004	<i>l</i>	循环和上下文堆栈处理
0x1000000	<i>M</i>	跟踪智能匹配解析
0x0000080	<i>m</i>	内存和SV分配
0x0000010	<i>o</i>	方法和重载解析
0x0000040	<i>P</i>	打印性能测试信息、源文件输入状态
0x0000001	<i>p</i>	词法分析和解析 (使用v, 显示解析栈)
0x0800000	<i>q</i>	静默, 目前仅抑制“EXECUTING”消息
0x0040000	<i>R</i>	包含转储变量的引用数 (例如使用-D时)
0x0000200	<i>r</i>	正则表达式解析和执行
0x0000002	<i>s</i>	堆栈快照 (使用v, 显示所有堆栈)
0x0020000	<i>T</i>	词法分析
0x0000008	<i>t</i>	跟踪轨迹执行
0x0001000	<i>U</i>	非官方, 用户探测 (仅用于未发布的私有用法)
0x0000800	<i>u</i>	对不安全的外部数据的污染检查
0x0100000	<i>v</i>	详细信息: 结合其他标志使用
0x0004000	<i>X</i>	便签簿分配 (“Xratchpad”)
0x0000400	<i>x</i>	语法树转存

这些标志都要求Perl可执行程序是为调试特别构建的。不过, 由于这不是默认设置, 所以除非你或系统管理员构建了这种特殊的调试版本的Perl, 否则根本不能使用-D开关。有关的详细信息参见Perl源文件目录中的INSTALL文件。不过可以简单地讲: 编译Perl时要为C编译器传入-DDEBUGGING。如果向你询问优化器和调试器标志时, 你包含了-g选项, 则会自动设置这个标志。如果你希望执行每一行Perl代码时打

印输出各行代码（就像`sh -x`对shell脚本提供的输出一样），则不能使用Perl的`-D`开关，而应如下所示：

```
# Bourne shell 语法
$ PERLDB_OPTS="NonStop=1 AutoTrace=1 frame=2" perl -dS program

# csh 语法
% (setenv PERLDB_OPTS "NonStop=1 AutoTrace=1 frame=2"; perl -dS program)
```

有关的详细内容和变化见第18章。

#### `-e PERLCODE`

可以用来输入一行或多行脚本。给定`-e`选项时，Perl不会在参数表中查找程序的文件名。处理`PERLCODE`参数时会认为末尾有一个换行符，所以可以给出多个`-e`命令来建立一个多行程序（一定要使用分号，就像文件中存储的正常程序一样）。尽管`-e`会对每个参数提供一个换行符，这并不表示必须使用多个`-e`开关；如果你的shell支持多行（如`sh`、`ksh`或`bash`），也可以传入一个多行脚本作为`-e`参数：

```
$ perl -e 'print "Howdy, ";print "@ARGV!\n";' world
Howdy, world!
```

对于`csh`，可能更好的做法是使用多个`-e`开关：

```
% perl -e 'print "Howdy, ";' \
      -e 'print "@ARGV!\n";' world
Howdy, world!
```

不论是隐式换行还是显式换行，在计算行数时都会统计在内，所以不论哪一种情况，第二个`print`都在`-e`脚本的第2行上。

#### `-E PERLCODE`

这个开关的表现类似于`-e`，不过它会显式启用（主编译单元中的）所有可选特性。对于v5.14 版本，这些特性包括`say`、`state`、`switch`和`unicode_strings`。参见第29章的`feature pragma`。

#### `-f` 在启动时禁止执行`$Config{sitelib}/sitecustomize.pl`。

Perl可以构建为启动时会默认执行`$Config{sitelib}/sitecustomize.pl`（在一个`BEGIN`块中）。这是一个钩子，允许系统管理员定制Perl的行为。例如，可以用来为`@INC`数组增加路径，使Perl能够找到不在标准位置上的模块。

实际上Perl会插入以下代码：

```
BEGIN {
    do {
        local $!;
        -f "$Config{sitelib}/sitecustomize.pl";
    } && do "$Config{sitelib}/sitecustomize.pl";
}
```

由于这实际上是一个`do`（而不是`require`），`sitecustomize.pl`不需要返回一个`true`值。

这个代码在`main`包中运行（在其自己的词法作用域中）。不过，如果脚本结束，并不会设置`$@`。

`$Config{sitelib}`的值也在C代码中确定，而不是从`Config.pm`读取，因为并没有加载这个`Config.pm`文件。

这个代码会很早执行。例如，对`@INC`做的任何改变都会出现在`perl -V`的输出中。当然，类似的，`END`块会很晚才执行。

要在运行时确定Perl中是否编译有这种功能，可以检查`$Config{usesitecustomize}`的值：

```
% perl -V:usesitecustomize
usesitecustomize='undef';
```

### `-F PATTERN`

通过`-a`开关完成自动分解时，可以用这个开关指定要分解的模式（否则`-a`开关没有任何意义）。这个模式必须用斜线（`/`）、双引号（`"`）或单引号（`'`）包围，否则会把它自动放在单引号中。要记住，要想通过shell传递引号，必须对引号加引号。至于如何加引号，这取决于具体的shell。

`-h` 汇总打印Perl命令行选项。

`-i`

### `-i EXTENSION`

指定`<>`构造处理的文件要“原地”编辑，为此要重命名输入文件、按原名打开输出文件，并选择这个输出文件作为`print`、`printf`和`write`调用的默认文件<sup>注3</sup>。`EXTENSION`用来修改原文件的名称建立一个备份副本。如果没有提供`EXTENSION`，就不会建立备份，而会覆盖当前文件。如果`EXTENSION`不包含`*`，这个字符串会追加到当前文件名的末尾。如果`EXTENSION`确实包含一个或多个`*`字符，则每个`*`会替换为当前所处理的文件名。按照Perl的说法，可以认为这是：

```
($backup = $extension) =~ s/\*/$file_name/g;
```

这允许你为备份文件使用一个前缀而不是后缀，或者甚至除了后缀以外还可以使用前缀：

```
% perl -pi'orig_*' -e 's/foo/bar/' xyx      # 备份到'orig_xyx'
```

甚至可以把原文件的备份副本放在另一个目录中（假设这个目录已经存在）：

```
% perl -pi'old/*.orig' -e 's/foo/bar/' xyx  # 备份到'old/xyx.orig'
```

以下每一对单行脚本都是等价的：

---

注3：从技术角度讲，这实际上并不是“原地”完成。这确实是同一个文件名，但是它是一个不同的物理文件。



```
% perl -pi -e 's/foo/bar/' xyx          # 覆盖当前文件
% perl -pi '*' -e 's/foo/bar/' xyx      # 覆盖当前文件

% perl -pi '.orig' -e 's/foo/bar/' xyx   # 备份到'xyx.orig'
% perl -pi '*.orig' -e 's/foo/bar/' xyx  # 备份到'xyx.orig'
```

从shell执行命令：

```
% perl -p -i.orig -e "s/foo/bar/;"
```

等同于使用程序：

```
#!/usr/bin/perl -pi.orig
s/foo/bar/;
```

这是以下写法的一种很方便的简写形式，相比之下，下面的代码则要长得多：

```
#!/usr/bin/perl
$extension = '.orig';
LINE: while (<>) {
    if ($ARGV ne $oldargv) {
        if ($extension !~ /\*/) {
            $backup = $ARGV . $extension;
        }
        else {
            ($backup = $extension) =~ s/\*/$ARGV/g;
        }
        unless (rename($ARGV, $backup)) {
            warn "cannot rename $ARGV to $backup: $!\n";
            close ARGV;
            next;
        }
        open(ARGVOUT, ">$ARGV");
        select(ARGVOUT);
        $oldargv = $ARGV;
    }
    s/foo/bar/;
}
continue {
    print; # 打印到原来的文件名
}
select(STDOUT);
```

这个长代码实际上等同于有*-i*开关的简单单行脚本，只不过*-i*形式不需要通过比较\$ARGV和\$oldargv来了解文件名何时改变。不过，它确实会对所选择的文件句柄使用ARGVOUT，并在循环之后恢复原来的STDOUT作为默认输出文件句柄。与上面的代码类似，Perl会创建备份文件，而不论输出是否真的改变。如果希望追加到每个文件末尾或者重置行编号，可以使用eof（不加小括号）定位每个输入文件的末尾，有关的例子见eof函数的描述。

对于一个给定的文件，如果Perl不能按EXTENSION指定的方式创建备份文件，它会发出一个警告，并继续处理列出的所有其他文件。

不能使用`-i`来创建目录或者从文件去除扩展名。也不能结合`~`使用这个开关来指示主目录—因为有些人喜欢用这个字符表示备份文件：

```
% perl -pi~ -e 's/foo/bar/' file1 file2 file3...
```

最后，即使命令行上没有给定任何文件名，`-i`开关也不会阻止Perl运行。如果发生这种情况，将不会建立备份，因为无法确定原文件，相应地，处理将从STDIN到STDOUT继续进行。

#### `-I DIRECTORY`

`-I`指定的目录追加到`@INC`前面，`@INC`中包含模块的搜索路径。与`use lib`类似，`-I`开关会隐含地增加平台特定的目录。有关的详细信息见第29章的`use lib`。

#### `-l`

#### `-l OCTNUM`

启用自动行结束处理。它有两个效果：首先，结合`-n`或`-p`使用时，它会自动使用`chomp`切断行结束符；其次，会设置`$\`为`OCTNUM`的值，使得所有`print`语句都在最后增加一个ASCII值为`OCTNUM`的行结束符。如果忽略`OCTNUM`，`-l`将`$\`设置为`$/`的当前值，通常是换行符。所以要将行截断为80列，可以写为：

```
% perl -lpe 'substr($_, 80) = ""'
```

注意赋值`$\ = $/`在处理这个开关时完成，所以如果`-l`开关后面是一个`-0`开关，输入记录分隔符可能与输出记录分隔符不同：

```
% gnufind / -print0 | perl -ln0e 'print "found $_" if -p'
```

这会把`$\`设置为换行符，随后再将`$/`设置为`null`字符（注意如果`0`直接跟在`-l`后面，它会被解释为`-l`开关的一部分。正是因为这个原因，我们在它们之间加上了`-n`开关）。

#### `-m`和`-M`

如果你执行了一个`use`，这些开关会加载一个`MODULE`，除非你指定了`-MODULE`而不是`MODULE`，在这种情况下它们会调用`no`。例如，`-Mstrict`就像是`use strict`，而`-M-strict`就像是`no strict`。

#### `-m MODULE`

执行脚本之前执行`use MODULE()`。

#### `-M MODULE`

执行脚本前执行`use MODULE`。这个命令是通过直接内插`-M`后的其余参数构成的，所以可以利用引号在模块名后面增加额外的代码。例如，`-M'MODULE qw(foo bar)'`。如果`-M`或`-m`后面的第一个字符是一个短横线（`-`），则`use`会替换为`no`。例如，要用它来确认运行Perl的最小版本号，可以使用`-Mv5.14`，确保至少运行v5.14或更高版本。

`-M MODULE=ARG1, ARG2...`

这是一个内置的小语法糖，表示可以用`-Mmodule=foo,bar`作为`-M'module qw(foo bar)'`的快捷方式。这样可以避免在导入符号时使用引号。

`-Mmodule=foo,bar`生成的具体代码是：

```
Use module split(/,/ , q{foo,bar})
```

需要说明，`=`形式消除了`-m`和`-M`之间的区别，不过最好使用大写形式来避免混淆。

只能从真正的Perl命令行调用来使用`-M`和`-m`开关，而不能作为`#!`行上的选项（嘿，如果你打算把它放在文件中，为什么不直接写等价的`use`或`no`呢）。

`-P` 由于可移植性问题v5.12中已经去除了这个开关。现在可以使用CPAN上的`Text::CPP`模块。

`-n` 让Perl假设你的脚本外围有以下循环，从而迭代处理文件名参数，就像`sed -n`或`awk`一样：

```
LINE:
while (<>) {
    ...                # 这里是你的脚本
}
```

可以在你的脚本中使用`LINE`作为一个循环标签，尽管你的文件中无法看到具体的标签。

需要说明，默认情况下并不打印行。如果要打印行，请参见`-p`。下面是一种很高效的做法，可以删除所有时间超过一周的文件：

```
find . -mtime +7 -print | perl -nle unlink
```

这比使用`find`程序的`-exec`开关速度更快，因为不用对找到的每一个文件名都开始一个进程。这种方法确实存在一个bug，可能会错误地处理路径名中的换行符，如果基于`-0`执行这个例子则可以修正这个问题。巧合的是，在这个隐式循环的前面或后面可以使用`BEGIN`和`END`块来捕获控制，就像`awk`中一样。

`-p` 使Perl假设你的脚本外围有以下循环，从而迭代处理文件名参数，就像`sed`一样：

```
LINE:
while (<>) {
    ...                # 这里是你的脚本
}
continue {
    (print) || die "-p destination: $!\n";
}
```

可以在你的脚本中使用`LINE`作为一个循环标签，尽管你的文件中无法看到具体的标签。



如果出于某种原因无法打开参数指定的文件，Perl会向你发出警告，并转向下一个文件。需要说明，这些行不会自动打印。打印中出现的错误会作为致命错误。不过这里又有一个巧合，可以在隐式循环前面或后面使用BEGIN和END块来捕获控制，就像awk中一样。

- s 有些开关在脚本名之后，但在文件名参数或“--”开关处理终止符之前，对于所有这些开关，-s会启用这些开关的基本开关解析。所找到的开关都会从@ARGV删除，并在Perl中设置一个与开关同名的变量。这里不允许天关绑定，因为允许有多字符开关。

只有当用一个-foo开关调用脚本时，以下脚本才会打印“true”。

```
#!/usr/bin/perl -s
if ($foo) { print "true\n" }
```

如果开关形如-xxx=yyy，\$xxx变量设置为该参数中等号后面的部分（这里是“yyy”）。当且仅当脚本用一个-foo=bar开关调用时，下面的脚本才会打印“true”。

```
#!/usr/bin/perl -s
if ($foo eq 'bar') { print "true\n" }
```

需要指出，类似--help的开关会创建变量\${-help}，这与strict refs不兼容。另外，在启用了警告的脚本上使用这个选项时，可能会生成有欺骗性的“used only once”（只使用一次）警告。

- S 令Perl使用PATH环境变量搜索脚本（除非脚本名包含目录分隔符）。

一般地，这个开关可以帮助在不支持#!的平台上模拟#!。很多平台上的shell与Bourne或C shell兼容，在这些平台上，可以使用以下脚本：

```
#!/usr/bin/perl
eval "exec /usr/bin/perl -S $0 $*"
    if $running_under_some_shell;
```

系统忽略第一行，并把脚本交给/bin/sh，它尝试将这个Perl脚本作为一个shell脚本来执行。这个shell将第二行作为一个正常的shell命令来执行，这会启动Perl解释器。在一些系统上，\$0并不总包含完整的路径名，所以-S告诉Perl要在必要的情况搜索脚本。Perl找到脚本后，它会解析各行，并将其忽略，因为变量\$running\_under\_some\_shell永远不为true。还有一个比\$\*更好的构造：\${1+"\$@"}，它会处理文件名中内嵌的空格等字符，不过如果脚本由csh解释，则不能工作。要启动sh而不是csh，有些系统必须把#!行替换为一个只包含一个冒号的行，Perl会将它礼貌地忽略。其他系统对此无法控制，而需要一个在csh、sh或perl下都能工作的“灵活”的构造，如下所示：

```
eval '(exit $?0)' && eval 'exec /usr/bin/perl -S $0 ${1+"$@"}'
    & eval 'exec /usr/bin/perl -S $0 $argv:q'
```

```
if 0;
```

没错，这确实很丑陋，不过采用这种方式工作的系统也同样如此<sup>注4</sup>。

在一些平台上，`-S`开关还会让Perl在搜索时为文件名追加后缀。例如，在Win32平台上，如果查找原文件名失败，而且文件名不是以`.bat`和`.cmd`后缀结束，就会追加这样的后缀。如果你的Perl构建时启用了调试特性，可以使用Perl的`-Dp`开关来查看搜索的进展。

如果提供的文件名包含目录分隔符（即使只是一个相对路径名，而不是绝对路径），另外如果该文件未找到，隐式追加文件扩展名的那些平台（非Unix）就会逐个地加上这些扩展名，然后再查找文件。

在类DOS平台上，如果脚本不包含目录分隔符，首先会在当前目录中搜索，然后才在PATH中搜索。在Unix平台上，由于安全性问题，有可能在没有显式请求的情况下意外地执行当前工作目录中的某个程序，所以会严格地在PATH上搜索脚本。

- t 类似于-T，不过污染检查会发出警告而不是致命错误。这些警告可以用`no warnings qw(taint)`正常地控制。

注意：这不能替代-T！这说明，只是在保护遗留代码的安全时可以把它作为一个临时的辅助开发手段。对于真正的生产代码，以及对于从头开始编写的新的安全代码，一定要用-T。

- T 强制打开“污染”检查，从而能进行检查。正常情况下，只有运行`setuid`或`setgid`时才会完成这些检查。一种好的做法是：对于为别人运行的程序（如CGI程序），都要显式打开这种检查。详细内容参见第20章。

出于安全性的原因，Perl必须很早就看到这个选项。通常这意味着它必须在命令行或`#!`行上较早出现。如果不够早，Perl会发出警告。

- u 令Perl在编译脚本之后转储内核。理论上讲，接下来可以使用`undump`程序（未提供）将这个内核转储转换为一个可执行文件。这会加快启动速度，但要以磁盘空间为代价（可以通过删除可执行文件中的符号（`strip`）来尽量减少磁盘空间）。如果希望在转储之前执行脚本的某一部分，可以使用Perl的`dump`操作符。需要说明，`undump`的可用性与平台相关，可能在Perl的某个特定移植版本中并不可用。这个开关已经被新的Perl-C代码生成器取代，后者有更好的可移植性（不过仍处于试验阶段）。

- U 允许Perl完成一些不安全的操作。目前“不安全”的操作包括作为超级用户在运行时解除目录链接，以及运行`setuid`程序时将致命污染检查转换为警告。注意必须启用警告，从而能真正生成污染检查警告。

---

注4： 我们故意用了这个词。



-v 打印Perl可执行文件的版本和补丁级别，以及额外的一些信息。

-V 汇总打印主要的Perl配置值和@INC当前值。

-V: NAME

将指定配置变量的值打印到STDOUT。NAME可能包含正则表达式字符，如“.”来匹配任意字符，或用“.\*”匹配任意可选的字符序列。

```
% perl -V:man.dir
man1dir='/usr/local/man/man1'
man3dir='/usr/local/man/man3'
```

```
% perl -V:'.*threads'
d_oldpthreads='undef'
use5005threads='define'
useithreads='undef'
usethreads='define'
```

如果请求一个不存在的配置变量，它的值会报告为“UNKNOWN”。配置信息可以使用Config模块从程序得到，不过散列下标不支持模式：

```
% perl -MConfig -le 'print $Config{man1dir}'
/usr/local/man/man1
```

有关的更多详细内容参见Config模块。

-w 对于只提到一次的变量，以及在设置之前就使用的标量值，这个开关会打印有关的警告。另外还会对重复定义的子例程、指向未定义文件句柄等发出警告。如果以只读方式打开文件句柄，但试图写该文件，也会发出警告。如果一个值要用作为数字但是看起来不像数字，如果把数组当作标量使用，如果子例程递归深度超过100层……诸如此类的问题都会发出警告。参见perldiag中所有标有“(W)”的条目。

这个开关只设置全局\$^W变量。它对词法警告没有任何作用，词法警告参见-W和-X开关。可以通过warnings（或no warnings）pragma启用或禁用特定的警告，如第29章所述。

-W 在整个程序中永久性地无条件地启用所有警告，即使局部使用no warnings或\$^W = 0禁用警告也没有影响。这包括通过use、require或do加载的所有文件。可以把它看作是Perl版本的lint(1)命令。

-x

-x DIRECTORY

告诉Perl抽取嵌入在一个消息中的脚本。前面的垃圾会被丢弃，直到遇到第一行以#!开头并包含字符串“perl”的代码。这一行上单词“perl”后面的所有有意义的开关都将应用。如果指定了一个目录名，Perl会切换到该目录，然后再运行脚本。-x开关只丢弃前面的垃圾，而不处理末尾的垃圾。如果有要忽略的末尾垃圾，脚本必须以\_\_END\_\_或\_\_DATA\_\_结尾（如果必要，脚本可以通过DATA文件句柄处理任何（或



所有) 末尾垃圾。理论上讲, 它甚至可以定位到文件起始位置, 并处理前面的垃圾)。

-X 永久地无条件地禁用所有警告, 它与-W标志的工作正好相反。

## 环境变量

除了可以利用各种开关显式地改变Perl的行为, 还可以设置环境变量影响Perl的底层行为。如何设置这些环境变量取决于具体的系统, 不过如果你使用*sh*, *ksh*或*bash*, 那么必须知道一个技巧: 可以为单个命令临时设置一个环境变量, 就好像它是一个有趣的开关。这个环境变量必须在命令前设置:

```
$ PATH='/bin:/usr/bin' perl myprogie
```

可以在*csh*和*tcsh*中用子shell做类似的工作:

```
% (setenv PATH "/bin:/usr/bin"; perl myprogie)
```

否则, 通常可以在主目录的某个文件中 (如*.cshrc*或*.profile*) 设置环境变量。在*csh*和*tcsh*上, 可以有:

```
% setenv PATH '/bin:/usr/bin'
```

在*sh*、*ksh*和*bash*上可以有:

```
$ PATH='/bin:/usr/bin'; export PATH
```

其他系统还有一种半永久地设置环境变量的方法。以下是Perl要注意的环境变量:

### HOME

如果调用*chdir*但没有提供参数, 则使用这个环境变量。

### LC\_ALL, LC\_CTYPE, LC\_COLLATE, LC\_NUMERIC, PERL\_BADLANG

这些环境变量控制Perl如何处理特定于某个自然语言的数据。有关内容参见*perllocale*的在线文档。

### LOGDIR

如果*chdir*没有参数而且未设置HOME, 则使用这个环境变量。

### PATH

执行子进程时使用, 如果使用了-S开关可以用这个环境变量查找程序。

### PERL5DB

加载调试器代码所用的命令。默认为:

```
BEGIN { require "perl5db.pl" }
```

关于这个变量的更多用法参见第18章。

#### PERL5DB\_THREADED

如果设置为true，指示调试器所调试的代码使用了线程。

#### PERL\_ALLOW\_NON\_IFS\_LSP（特别针对Win32移植版本）

如果设置为1，则允许使用非IFS兼容的LSP。Perl通常会搜索IFS兼容的LSP，因为用Windows套接字模拟真正的文件句柄时要求IFS兼容。不过，如果有一个防火墙（如*McAfee Guardian*），这可能会带来问题，它要求所有应用使用其LSP，而且不是IFS兼容的，因为显然正常情况下Perl会避免使用这种LSP。

将这个环境变量设置为1，意味着Perl只会使用编目中模拟的第一个合适的LSP，这会满足*McAfee Guardian*的要求（在这个特定的情况下，Perl也能工作，因为*McAfee Guardian*的LSP实际上会做另外一些处理，以允许那些要求IFS兼容性的应用也能工作）。

#### PERL\_DEBUG\_MSTATS

只有编译Perl时包含Perl发布版本提供的malloc（也就是说，如果perl -V:d\_mymalloc为define），这个环境变量才有意义。如果设置了这个环境变量，会转储执行后的内存统计信息。如果设置为一个大于1的整数，还会转储编译后的内存统计信息。

#### PERL\_DESTRUCT\_LEVEL

只有当构建Perl可执行程序时指定了-DDEBUGGING开关，这个环境变量才有意义。这个变量会控制对象及其他引用的全局撤销行为。有关的更多信息参见perlhacktips中的“PERL\_DESTRUCT\_LEVEL”。

#### PERL\_DL\_NONLAZY

如果这个环境变量设置为1，会让Perl在加载动态库时解析所有未定义的符号。默认行为是真正使用一个符号时才进行解析。完成扩展包测试时设置这个变量很有用，因为这样可以确保拼写有误的函数名一定会导致一个错误，即使测试用例并没有真正调用这个函数。

#### PERL\_ENCODING

不要使用这个环境变量。它依赖于encoding pragma，而这个pragma不能正常工作。

#### PERL\_HASH\_SEED

（从v5.8.1开始）这个环境变量用来随机化Perl的内部散列函数。要模拟5.8.1以前的行为，可以将它设置为一个整数（0代表与v5.8.0相同的顺序）。“5.8.1以前”有很多含义，其中包括多次运行Perl时，散列值总有相同的顺序。

默认情况下，大多数散列都会按v5.8.0同样的顺序返回元素。如果一个散列键插入过程中检测到有问题的数据，则会切换为另一个候选的随机散列种子。

默认行为是随机化，除非设置了`PERL_HASH_SEED`。如果Perl编译时指定了`-DUSE_HASH_SEED_EXPLICIT`开关，默认行为则是不随机化，除非设置了`PERL_HASH_SEED`。

如果`PERL_HASH_SEED`未设置，或者设置为一个非数值字符串，Perl会使用操作系统和库提供的伪随机种子。

一定要注意，散列种子是敏感信息。散列随机化是为了保护Perl代码不受到本地和远程攻击。通过手动地设置一个散列种子，这种保护可能就会部分或完全丧失。

有关的更多信息参见*perlsec*中的“Algorithmic Complexity Attacks”（算法复杂性攻击）以及*perlrun*中的“ENVIRONMENT”（环境）。

#### PERL\_HASH\_SEED\_DEBUG

（从v5.8.1）这个环境变量设置为1时，会（向STDERR）显示执行开始时散列种子的值。结合`PERL_HASH_SEED`“参见*perlrun*中的‘xPERL\_HASH\_SEED’”可以帮助调试散列随机性导致的不确定行为。

注意散列种子是敏感信息：如果知道了散列种子，就能对Perl代码发动拒绝服务攻击，甚至可以远程发动这种攻击；有关的更多信息参见*perlsec*中的“Algorithmic Complexity Attacks”（算法复杂性攻击）。不要向不需要知道的人暴露散列种子。参见Hash::Util的hash\_seed()。

#### PERL\_MEM\_LOG

如果将Perl配置为`-Accflags=-DPERL_MEM_LOG`，设置环境变量`PERL_MEM_LOG`将启用记录调试消息。这个环境变量的值形式为`number[m][s][t]`，这里`number`是你想写入的文件描述符编号（默认为2），后面的字母组合指定你想要得到有关内存（m）和/或（s）v的信息，还可以有时间戳信息（t）。例如，`PERL_MEM_LOG=1mst`将把所有信息记入stdout。还可以采用多种方式写至其他打开的文件描述符：

```
bash$ 3>foo3 PERL_MEM_LOG=3m perl ...
```

#### PERL\_ROOT（针对VMS移植版本）

仅用于VMS，这个转换会隐藏包含Perl的根逻辑名和对应@INC路径的逻辑设备。在VMS上还有一些影响Perl的逻辑名，包括`PERLSHR`、`PERL_ENV_TABLES`和`SYS$TIMEZONE_DIFFERENTIAL`，不过这些都是可选的，在*perlvms*中有进一步的讨论，另外参见Perl源代码版本的`README.vms`。

#### PERL\_SIGNALS

在v5.8.1及以后版本中，如果这个环境变量设置为`unsafe`，就会恢复Perl-5.8.0以前版本的行为（即时但不安全的信号）。如果设置为`safe`，则使用安全（或延迟）的信号。参见*perlipc*中的“Deferred Signals (Safe Signals)”（延迟信号（安全信号））。



## PERL5SHELL (只适用Microsoft移植版本)

可以设置为另一个候选的shell, Perl在内部必须使用这个shell通过反引号或system执行命令。WinNT上默认为`cmd.exe /x/c`, Win95上默认为`command.com /c`。Perl认为这个值是用空格分隔的。在所有需要保护的字符(如空格或反斜线)前面要加上一个反斜线。

需要说明, 对此Perl没有使用COMSPEC, 因为对于用户来说, COMSPEC的可变性很大, 这会导致移植问题。另外, Perl可能使用一个不适合交互使用的shell, 将COMSPEC设置为这样一个shell会影响其他程序的正常工作(通常要查看COMSPEC来找到适合交互使用的shell)。

## PERL5LIB

这是一个用冒号分隔<sup>注5</sup>的目录列表, 在查看标准库和当前目录之前, 要在这些目录中查找Perl库文件。会自动包含特定位置中特定体系结构的目录(如果这些目录存在)。如果PERL5LIB未定义, 则使用PERLLIB, 以保证与较早版本的向后兼容性。

运行污染检查时(可能由于程序在运行setuid或setgid, 或者使用了-T开关), 这两个库变量都不会使用。这些程序必须使用一个显式的lib pragma。

## PERL5OPT

默认的命令行开关。会认为每一个Perl命令行上都有这个变量中的开关。只允许-[DIMUdmw]开关。

运行污染检查时(可能由于程序在运行setuid或setgid, 或者使用了-T开关), 这个变量会被忽略。如果PERL5OPT以-T开头, 则启用污染检查, 这会导致所有后续的选项被忽略。

## PERLIO

这是一个用空格(或冒号)分隔的PerlIO层列表。如果将Perl构建为使用PerlIO系统完成IO(默认), 这些层会影响Perl的IO。

传统做法是层名以一个冒号开头(例如:perlio), 以强调它们与变量“属性”的相似性。不过解析层规范字符串的代码(也用于解码PERLIO环境变量)会把这个冒号处理为一个分隔符。

如果PERLIO未设置为空, 这等价于你的平台的层默认设置。例如, 类Unix系统上的:unix:perlio, 以及Windows和其他类DOS系统上的:unix:crlf。

这个列表将成为所有Perl的默认IO列表。相应地, 这个列表中只能出现内置的层, 因为外部的层(如:encoding(LAYER))需要IO来完成加载。如果要增加外部编码作为默认编码, 有关的内容参见第29章的open pragma。

---

注5: 在Unix及其派生系统中确实是用冒号分隔。在Microsoft系统上则是用分号分隔。

可以在PerlIO环境变量中包含一些层，下面简要地做一个总结：

#### `:bytes`

这是一个伪层，会关闭下面的层的`:utf8`标志。一般这个层不太可能单独用在全局PerlIO环境变量中。你可能会考虑使用`:crlf:bytes`或`:perlio:bytes`。

#### `:crlf`

这一层会完成CRLF到“\n”转换，按MS-DOS和类似操作系统的方式区分“文本”和“二进制”文件（不过目前在处理Control-Z作为文件末尾标志方面，它并没有模仿MS-DOS的做法）。

#### `:mmap`

这一层实现文件“读”，使用`mmap`使（整个）文件出现在进程的地址空间中，然后将它用作为PerlIO的“缓冲区”。

#### `:perlio`

这是对“类stdio”缓冲的一个重新实现，将它写作为一个PerlIO“层”。因此，它会调用下面的层来完成其操作（通常是`:unix`）。

#### `:pop`

这是一个试验性的伪层，会删除最顶层。使用时要像使用炸药一样格外小心。

#### `:raw`

这是一个管理其他层的伪层。应用`:raw`层等价于调用`binmode($fh)`。它使流原样传递每一个字节，而不做任何解码。具体地，CRLF转换和本地化环境变量的`:utf8`变换都会禁用。

与Perl以前的版本不同，`:raw`不完全是`:crlf`的逆操作。可能影响流二进制性质的其他层也会删除或禁用。

#### `:stdio`

这一层通过包装系统的ANSI C“stdio”库调用来提供PerlIO接口。这一层同时提供缓冲和IO。需要说明，`:stdio`层不做CRLF转换，尽管这是平台的正常行为。要完成CRLF转换，还需要在这一层上面加一个`:crlf`层。

#### `:unix`

这是调用`read`、`write`、`lseek`等的底层层。

#### `:utf8`

这是一个伪层，在下一层启用一个标志来告诉Perl输出应当采用UTF-8格式，而且认为输入必然采用合法的UTF-8格式。它不会检查合法性，因此作为输入时一定要小心处理。如果对输入使用这个`:utf8`层，一定要启用UTF-8警告（最好是致命警告）。否则，应当在读取UTF-8编码数据时使用`:encoding(UTF-8)`。

:win32

这是一个试验性的层，在Win32平台上，这个层使用原生“句柄”IO而不是一个类Unix的数值文件描述符层。已经知道v5.14版本中这个层还有bug。

默认(layer)的设置应当能在所有平台上都能得到可接受的结果。对于Unix平台，这等于“unix perlio”或“stdio”。如果系统库可以提供对缓冲区最快的访问，设置配置时则倾向于“stdio”实现；不过使用“unix perlio”实现更为常见。

在Win32上，v5.14版本中的默认设置是“unix crlf”。Win32的“stdio”对于Perl IO有很多bug，或者更确切地讲，存在一些有问题的特性，这些特性有些依赖于C编译器的版本和开发商。使用我们自己的crlf层作为缓冲区可以避免这些问题，并更为一致。crlf层提供了CRLF“\n”转换以及缓冲处理。

在Win32上，Perl v5.14使用unix作为最底层，所以仍使用C编译器的数值文件描述符例程。还有一个试验性的原生win32层，有望将来增强，并最终成为Win32的默认层。

Perl在污染模式中运行时，会完全忽略PERLIO环境变量。

#### PERLIO\_DEBUG

如果设置为一个文件或设备的名字，PerlIO子系统的某些操作会记入该文件（以追加模式打开）。在Unix中，通常用法如下：

```
% env PERLIO_DEBUG=/dev/tty perl script ...
```

而在Win32中，相应的用法为：

```
> set PERLIO_DEBUG=CON
> perl script ...
```

对于setuid脚本和用-T运行的脚本，会禁用这个功能。

#### PERLLIB

这是一个冒号分隔的目录列表，查看标准库和当前目录前会在这里查找Perl库文件。如果定义了PERL5LIB，则不使用PERLLIB。

#### PERL\_UNICODE

等价于-C命令行开关。需要说明，这不是一个布尔变量。将它设置为“1”并不是“启用Unicode”的正确做法。不过，可以使用“0”来“禁用Unicode”（或者，启动Perl之前在shell中取消PERL\_UNICODE）。

在涉及文本而不是二进制的大多数情况下，将这个变量设置为“AS”通常很有用：它会隐式地将@ARGV从UTF-8解码，并使用binmode将STDIN、STDOUT和STDERR句柄读取到内置:utf8层。如果这些将是UTF-8文本而不只是二进制字节流，就可以使用这个变量。将这个变量设置为“ASD”对于某些情况可能更为有用，不过因为它还会改变所有文件句柄的默认编码（从二进制到:utf8），这会破坏很多老程序，因为这些老程



序原来假设为二进制流（或在Windows上假设为文本流），因此没有考虑自行调用binmode。Unix程序在这方面可谓臭名昭著。因此，“D”设置最好只用于暂时运行。

由于内置:u t f 8层默认情况下不会产生异常，甚至不会对输入流中不合法的UTF-8发出警告，要得到正确的行为，如果在输入流上使用:u t f 8层，还必须启用“utf8”警告。在命令行上，可以使用-Mwarnings=utf8启用警告，或者使用-Mwarnings=FATAL,utf8启用异常。这对应于在程序中使用use warnings "utf8"和use warnings FATAL => "utf8"。参见第6章的“获取Unicode数据”一节。

#### SYS\$LOGIN（针对VMS移植版本）

如果chdir没有参数，而且没有设置HOME和LOGDIR，就会使用这个变量。

除此以外，Perl本身不使用其他环境变量，除非要让正在执行的程序以及该程序启动的所有子进程使用那些环境变量。有些模块（标准模块或非标准模块）可能还会关心其他环境变量。例如，re pragma使用PERL\_RE\_TC和PERL\_RE\_COLORS，Cwd模块使用PWD，CGI模块使用HTTP守护进程（即Web服务器）设置的很多环境变量来向CGI脚本传递信息。

运行setuid的程序在做其他工作之前可以执行以下代码行，以保证人们诚实：

```
$ENV{PATH} = '/bin:/usr/bin';    # 或都你需要的任何路径
$ENV{SHELL} = '/bin/sh' if exists $ENV{SHELL};
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)};
```

有关的详细内容参见第20章。

# Perl调试器

首先要问一句，你用过warnings pragma吗？

如果调用Perl时提供了-d开关，你的程序就会在Perl调试器中运行。它就像一个交互式Perl环境，会提示你输入调试器命令，允许你检查源代码、设置断点、转储函数调用堆栈、修改变量值等。调试器无法识别的命令会作为当前所调试的代码包中的Perl代码（使用eval）直接执行（调试器使用DB包维护自己的状态信息，以避免影响你的状态）。这非常方便，以至于人们通常启动调试器只是为了交互式地测试Perl构造。对于这种情况，让Perl调试哪个程序并不重要，所以我们会选择一个没有太大意义的程序：

```
% perl -de 42
```

在Perl中，调试器并不是与所调试程序完全分离的一个程序，这与一般的编程环境不同。实际上，-d标志告诉编译器将源码信息插入到将交给解释器的解析树中。这说明，你的代码首先必须正确地编译，调试器才能对它进行调试。如果编译成功，解释器会预加载一个特殊的Perl库文件，其中包含调试器本身。

```
% perl -d /path/to/program
```

这个程序在第一个运行时可执行语句前停止（关于编译时语句，参见下一节“使用调试器”），并要求你输入一个调试器命令。不论调试器何时停止并向你显示一行代码，它显示的都是将要执行的一行代码，而不是刚刚执行过的那行代码。

调试器遇到一行代码时，它首先检查断点，打印这行代码（如果调试器处于跟踪模式），完成所需的动作（用本章后面“调试器命令”一节介绍的a命令创建），最后提示用户是否有断点，或者调试器是否处于单步模式。如果不是，则正常地执行这一行，并继续处理下一行。

# 使用调试器

调试器提示符如下所示：

```
DB<8>
```

或者甚至是：

```
DB<<17>>
```

这里的数字显示了已经执行了多少个命令。类`ssh`的历史机制还允许按数字访问之前执行过的命令。例如，`!17`会重复执行命令17。尖括号数指示了调试器的深度。例如，如果已经有一个断点，然后要打印一个函数调用的结果，而该函数本身也有一个断点，你就会得到另外一组尖括号。

如果想输入一个多行的命令，如包含多条语句的一个子例程定义，可以用一个反斜线对调试器命令末尾常有的换行符转义。下面给出一个例子：

```
DB<1> for (1..3) { \
    cont: print "ok\n"; \
    cont: }
ok
ok
ok
```

假想你对你的一个小程序启动调试器（假设这个小程序名为`camel_flea`），一旦遇到名为`infested`的函数就停止。如下所示：

```
% perl -d camel_flea
Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(camel_flea:2):    pests('bactrian', 4);
DB<1>
```

调试器会在第一个运行时可执行语句前停止你的程序（编译时语句将在下面介绍），并要求你输入一个命令。重申一次，只要调试器停下来向你显示一行代码，它显示的是将要执行的代码，而不是它刚刚执行过的那行代码。所显示的那行代码看起来与源文件中并不完全一样，特别是如果你通过某种预处理器运行这个程序，所看到的则是经过预处理的代码。

现在，你希望一旦到达`infested`函数，程序就停止，所以要在这里建立一个断点，如下所示：

```
DB<1> b infested
DB<2> c
```



现在调试器会继续，直到它到达这个函数，到达该函数时将输出：

```
main::infested(camel_flea:8):      my $bugs = int rand(3);
```

要查看断点附近的一个源代码“窗口”，可以使用w命令：

```
DB<2> w
5      }
6
7      sub infested {
8==>b    my $bugs = int rand(3);
9:        our $Master;
10:       contaminate($Master);
11:       warn "needs wash"
12:       if $Master && $Master->isa("Human");
13
14:       print "got $bugs\n";
DB<2>
```

看到==>标记时，当前行是行8，从这里的b可以知道这一行有一个断点。如果已经设置了一个动作，这里还会有一个a。如果行号里有冒号，说明这些行是可以中断的；其余的行则不能中断。

要看谁是调用者，谁是被调用者，可以用T命令请求一个栈回溯：

```
DB<2> T
$ = main::infested called from file `Ambulation.pm' line 4
@ = Ambulation::legs(1, 2, 3, 4) called from file `camel_flea' line 5
. = main::pests('bactrian', 4) called from file `camel_flea' line 2
```

最前面的字符（\$、@或.）分别指出这个函数是在一个标量上下文、列表上下文还是void上下文中调用。这里有3行，因为运行栈回溯时有3层函数调用。各行的含义分别为：

- 第一行指出运行栈回溯时正处在函数main::infested中。它指出这个函数从Ambulation.pm文件的第4行调用，并且是在标量上下文中调用。它还显示了调用这个函数时没有任何参数，这说明它是作为&infested调用的，而不是通常的infested()。
- 第二行显示函数Ambulation::legs在列表上下文中调用，具体是从camel\_flea文件的第5行调用，并有4个参数。
- 第三行显示main::pests在void上下文中调用，这是从camel\_flea文件的第2行调用的。

如果有编译阶段可执行语句，如BEGIN和CHECK块中的代码或use语句，正常情况下这些语句不会被调试器中断，不过调试器可以中断requires和INIT块，因为它们在转换到运行阶段后才发生（参见第16章）。通过在PERLDB\_OPTS中设置AutoTrace选项，可以跟踪编译阶段语句。

可以从Perl程序本身对Perl调试器施加一些控制。例如，只要在调试器下运行一个特定的程序，就在某个子例程设置一个自动断点。不过，可以从你自己的Perl代码使用以下语句把控制交回给调试器，即使调试器没有运行，这也没有危害：

```
$DB::single = 1;
```

如果把\$DB::single设置为2，这等价于n命令，值为1时则类似于s命令。\$DB::trace变量应当设置为1来模拟t命令。

调试模块的另一种方法是在加载时设置断点：

```
DB<7> b load c:/perl/lib/Carp.pm
Will stop on load of 'c:/perl/lib/Carp.pm'.
```

然后使用R命令重启调试器。为了更细粒度地控制，可以使用b compile subname，在特定子例程完成编译之后尽快停止。

## 调试器命令

在调试器中键入命令时，不需要末尾加分号。可以使用一个反斜线让行延续（不过这只适用于调试器中）。

由于调试器使用eval执行命令，一旦命令返回，my和local设置就会消失。如果一个调试器命令与程序中的某个函数冲突，只需要在函数调用前加上某个符号，使它看起来不像一个调试器命令，如一个前导;或一个+。

如果一个调试器内置命令的输出太长，无法在屏幕上完全显示，只需要在命令前面加一个前导管道符号，这样就能通过分页工具来运行：

```
DB<1> |h
```

调试器有很多命令，我们把这些命令（有些随意地）分为以下几类：步进和运行、断点、跟踪、显示、定位代码、自动命令执行，当然还有杂项。

可能最重要的命令就是h，这个命令用来提供帮助。如果在调试器提示窗口键入h h，你会得到一个精简的帮助列表，专门设计为可以在一屏上显示。如果键入h COMMAND，会得到关于这个调试器命令COMMAND的帮助。

## 步进和运行

调试器逐行地步进执行程序。下面的命令允许你控制要跳过哪些代码，另外在哪里停止。

s [EXPR]

s调试器命令单步执行程序。也就是说，调试器会执行程序的下一行代码，直到到达

另一条语句，必要时会深入到子例程调用中。如果要执行的下一行涉及一个函数调用，调试器会在该函数中的第一行停止。如果提供了一个`EXPR`，其中包含一些函数调用，这些函数也会单步执行。

**n** [`EXPR`]

`n`命令执行子例程，但不单步执行，直到到达同一层（或更高层）下一条语句的开头。如果指供了一个`EXPR`，其中包含一些函数调用，将执行这些函数，并在每个语句前停止。

**<ENTER>**

如果在调试器提示窗口敲回车，会重复前一个`n`或`s`命令。

**.** 命令返回最后执行的那行代码的内部调试器指针，并打印那一行代码。

**r** 这个命令会继续执行，直到当前执行的子例程返回。如果设置了`PrintRet`选项（默认值），还会显示返回值。

## 断点

**b**

**b** `LINE`

**b** `CONDITION`

**b** `LINE CONDITION`

**b** `SUBNAME`

**b** `SUBNAME CONDITION`

**b** `postpone SUBNAME`

**b** `postpone SUBNAME CONDITION`

**b** `compile SUBNAME`

**b** `load FILENAME`

`b`调试器命令在`LINE`前设置一个断点，告诉调试器在这一点停止程序，以便你进行检查。如果忽略`LINE`，则在将要执行的那一行设置断点。如果指定了`CONDITION`，每次到达该语句时都要完成一个计算：如果`CONDITION`为ture则触发一个断点。只能在开始一个可执行语句的行上设置断点。要注意条件不使用`if`：

```
b 237 $x > 30
b 237 ++$count237 < 11
b 33 /pattern/i
```

`b SUBNAME`形式会在指定的子例程的第一行前设置一个（可能有条件的）断点。`SUBNAME`可以是一个包含代码引用的变量，如果是这样，则不支持`CONDITION`。

在尚未编译的代码上设置断点有很多方法。`b postpone`形式会在编译`SUBNAME`之后该子例程的第一行上设置一个（可能有条件的）断点。



**b compile**形式会在编译**SUBNAME**之后将要执行的第一条语句上设置一个断点。需要说明，与**postpone**形式不同，这个语句在当前子例程之外，因为这个子例程还未调用，只是完成了编译。

**b load**形式会在文件的每一个执行行上设置一个断点。**FILENAME**应当是一个完全路径名（如**%INC**中的值）。

**d**

**d LINE**

这个命令删除**LINE**上的断点；如果省略**LINE**，则删除将要执行的下一行上的断点。

**D** 这个命令删除所有断点。

**L** 这个命令列出所有断点和动作。

**c**

**c LINE**

这个命令继续执行，可能会在指定的**LINE**上插入一个一次性断点。

## 跟踪

**T** 这个命令生成一个栈回溯。

**t**

**t EXPR**

这个命令会切换跟踪模式，执行时会打印程序的每一行。参见这一章后面讨论的**AutoTrace**选项。如果提供了**EXPR**，调试器会跟踪其执行。参见后面“不被注意的执行”一节。

**W**

**W EXPR**

这个命令将**EXPR**增加为一个全局监视表达式（**watch expression**）。监视表达式的值改变时会触发一个断点。如果没有提供**EXPR**，所有监视表达式都将被删除。

## 显示

Perl的调试器有很多显示命令，程序在一个断点停止时可以用这些命令检查数据结构。

**p**

**p EXPR**

这个命令等同于在当前包中执行**print DB::OUT EXPR**。特别要注意，由于这只是Perl自己的**print**函数，嵌套的数据结构和对象不会显示，要想显示这些嵌套数据结构，需要使用**x**命令。**DB::OUT**句柄总是打印输出到你的终端（或者可能是一个编辑器窗口），而不论标准输出被重定向到哪里。

x

x *EXPR*

x命令在列表上下文计算其表达式，并用很漂亮的方式显示结果。也就是说，嵌套的数据结构会递归地打印，而且无法查看的字符也会得到适当的编码。

v

v *PKG*

v *PKG VARS*

这个命令使用美化打印显示指定*PKG*包中（默认为main包）的所有变量（或者如果指定了*VARs*，则显示指定的这些变量）。散列会显示其键和值，控制字符会以可识别的方式地显示，嵌套数据结构也以一种可读的方式打印，如此等等。这类似于在每个可用的变量上调用x命令，不过x命令还可以用于词法作用域变量。另外，这里键入标识符时无需类型指示符（如\$或@），如下所示：

```
V Pet::Camel SPOT FIDO
```

*VARs*中也可以不指定变量名，而使用~*PATTERN*或!*PATTERN*来打印名字与指定模式匹配或不匹配的现有变量。

x

x *VARs*

这个命令等同于v *CURRENTPACKAGE*，当前行要编译到*CURRENTPACKAGE*指定的包中。

h

h -*NUMBER*

这个命令显示最后*NUMBER*个命令。历史中只存储超过1个字符的命令（否则，历史中存储的大多数都将是s或n）。如果忽略*NUMBER*，会列出所有命令。

## 定位代码

在调试器中，可以用以下命令抽取和显示程序的某些部分。

l

l *LINE*

l *SUBNAME*

l *MIN+INCR*

l *MIN-MAX*

l命令列出程序的下面几行代码，或者如果提供了*LINE*，则显示指定的*LINE*行代码，或者显示*SUBNAME*子例程或代码引用的前几行代码。

l *MIN+INCR*形式会列出从*MIN*开始的*INCR*+1行。l *MIN-MAX*形式列出从*MIN*到*MAX*的各行代码。

- 这个命令列出程序的前面几行代码。

**w** [*LINE*]

列出给定源代码行*LINE*周围的一个窗口（几行代码），如果没有指定*LINE*，则列出当前行周围的一个窗口。

**f** *FILENAME*

这个命令允许你查看一个不同的命令或eval语句。如果*FILENAME*不像%INC中那样是一个完全路径名，将把它解释为一个正则表达式，用来查找你表示的文件名。

**/***PATTERN***/**

这个命令在程序中前向搜索*PATTERN*，最后的/可选。整个*PATTERN*也是可选的，如果忽略这个*PATTERN*，则重复之前的搜索。

**?PATTERN?**

这个命令反向搜索*PATTERN*，最后的?可选。如果忽略*PATTERN*，则重复之前的搜索。

**S**

**S** *PATTERN*

**S** **!***PATTERN*

S命令列出与*PATTERN*匹配（或者如果有!，则是与*PATTERN*不匹配）的那些子例程名。如果没有提供*PATTERN*，则列出所有子例程。

## 动作和命令执行

在调试器中，可以指定在特定时刻执行的动作。还可以启动外部程序。

**a**

**a** *COMMAND*

**a** *LINE*

**a** *LINE* *COMMAND*

这个命令设置*LINE*执行前将完成的动作，或者如果忽略了*LINE*，则是在当前行之前要完成的动作。例如，利用下面的命令，每次到达53行时都会打印\$foo：

```
a 53 print "DB FOUND $foo\n"
```

如果没有指定*COMMAND*，则删除指定*LINE*上的动作。如果既没有*LINE*也没有*COMMAND*，则删除当前行上的动作。

**A**      A调试器命令删除所有动作。

**<**

**<** **?**

**<** *EXPR*



<< *EXPR*

< *EXPR*形式指定每个调试器提示符前要计算的Perl表达式。可以用<< *EXPR*形式增加另一个表达式，用< ?列出这些表达式，还可以用一个<删除所有表达式。

>

> ?

> *EXPR*

>> *EXPR*

>命令的行为与相应的<命令很相似，不过会在调试器提示符之后（而不是之前）执行。

{

{ ?

{ *COMMAND*

{{ *COMMAND*

{调试器命令的表现与<很类似，不过它指定调试器提示符之前执行的一个调试器命令，而不是Perl表达式。如果不小心输入一个代码块，会发出一个警告。如果你确实想指定一个代码块，可以使用;{ ... }或者甚至do { ... }。

!

! *NUMBER*

! -*NUMBER*

!*PATTERN*

单独的!会重复之前的命令。*NUMBER*指定要执行历史中的哪个命令。例如，! 3会执行调试器中键入的第3条命令。如果*NUMBER*前有一个负号，命令则要倒数：! -3会执行倒数第3条命令。如果提供了一个*PATTERN*（无斜线）而不是*NUMBER*，就会执行最后一个以*PATTERN*开头的命令。参见recallCommand调试器选项。

!! *CMD*

这个调试器命令在一个子进程中运行外部命令*CMD*，从DB::IN读，并写至DB::OUT。参见shellBang调试器选项。这个命令使用\$ENV{SHELL}中指定的shell，有时可能影响状态、信号和核心转存信息的正确解释。如果想从命令得到一致的退出值，要把\$ENV{SHELL}设置为/bin/sh。

|

! *NUMBER*

! -*NUMBER*

!*PATTERN*

|*DBCMD*命令运行调试器命令*DBCMD*，将DB::OUT管道输出到\$ENV{PAGER}。这通常与那些可能产生很长输出的命令结合使用，如：

```
DB<1> |V main
```

注意，这只适用于调试器命令，而不适用于在shell中键入的命令。如果想把外部命令`who`的输出通过分页工具显示，可以如下完成：

```
DB<1> !!who | more
```

`||PERLCMD`命令类似于`|DBCMD`，不过会临时选择`DB::OUT`，所以如果调用`print`、`printf`或`write`的命令没有提供文件句柄，也会沿这个管道发送。例如，如果一个函数通过调用`print`可能生成大量输出，可以使用这个命令而不是之前的命令来对这个输出分页显示：

```
DB<1> sub saywho { print "Users: ", `who` }  
DB<2> ||saywho()
```

## 杂项命令

`q`和`^D`

这些命令会退出调试器。这是推荐的退出方式，不过输入两次`exit`有时也是可以的。如果你想跳出到程序末尾但仍保留在调试器中，可以将`inhibit_exit`选项设置为0。如果想要单步执行全局撤销，可能还需要将`$DB::finished`设置为0。

**R** 通过`exec`执行一个新会话来重启调试器。调试器会努力跨会话维护历史，不过有些内部设置和命令行选项可能会丢失。目前会保留下面的设置：历史、断点、动作、调试器选项以及Perl命令行选项`-w`，`-I`和`-e`。

`=`

`= ALIAS`

`= ALIAS VALUE`

这个命令打印`ALIAS`的当前值（如果没有给定`VALUE`）。如果有`VALUE`，它会定义一个名为`ALIAS`的新的调试器命令。如果`ALIAS`和`VALUE`都忽略，则会列出当前的所有别名。例如：

```
= quit q
```

所有`ALIAS`都应当是简单标识符，或者可以转换为一个简单标识符。通过在`%DB::aliases`中直接增加你自己的记录，可以建立更复杂的别名设置。参见下一节“调试器定制”。

`man`

`man MANPAGE`

这个命令调用系统的默认文档阅读器查看给定页面，如果忽略`MANPAGE`，则显示阅读器本身。如果这个阅读器是`man`，将使用当前的`%Config`信息调用这个阅读器。必要时会为你自动提供“perl”前缀，这就允许你在调试器中直接输入`man debug`和`man op`。

在通常不提供`man`工具的系统上，调试器会调用`perldoc`；如果你想改变这个行为，可以将`$DB::doccmd`设置为你喜欢的任何阅读器。可能在一个`rc`文件中设置，或者通过直接赋值来设置。

0

0 *OPTION* ...

0 *OPTION?* ...

0 *OPTION=VALUE* ...

0命令允许你管理调试器选项，本章后面的“调试器选项”一节会列出这些选项。

0 *OPTION*形式将所列的各个调试器选项设置为1。如果*OPTION*后面有一个问号，会显示它的当前值。0 *OPTION=VALUE*形式设置值；如果*VALUE*内部包含空白符，则要加强引号。例如，可以设置0 *pager="less -MQeicsNfr"*，从而结合那些特定标志使用`less`。可以使用单引号或双引号，不过如果使用了引号，就必须对嵌入的同类引号转义。如果引号前面有反斜线，但这些反斜线并不表示对引号本身转义，那么还需要对这些反斜线进行转义。换句话说，只要遵循单引号规则，而不论具体使用什么引号。调试器会显示刚设置的选项值作为响应，而且输出总使用单引号记法：

```
DB<1> 0 OPTION='this isn\'t bad'
      OPTION = 'this isn\'t bad'

DB<2> 0 OPTION="She said, \"Isn't it?\""
      OPTION = 'She said, "Isn\'t it?"'
```

出于历史原因，`=VALUE`是可选的，不过只在安全的情况下才默认为1。也就是说，主要用于布尔选项。最好用`=`赋一个特定的*VALUE*。*OPTION*可以缩写，不过除非你有意想增加神秘感，否则最好不要这么做。可以同时设置多个选项。调试器选项列表参见本章后面的“调试器选项”一节。

## 调试器定制

调试器已经包含足够多的配置钩子，可能根本不用你自己修改。你可以使用0命令在调试器中改变调试器行为，也可以通过`PERLDB_OPTS`环境变量从命令行修改，还可以通过运行存储在`rc`文件中的预置命令来改变调试器行为。

## 编辑器提供的调试支持

调试器的命令行历史机制并不像很多shell那样提供命令行编辑：不能用`^p`访问前面的行，也不能用`^a`移至行的起始位置，但你可以用shell用户熟悉的感叹号语法执行前面的行。不过，如果从CPAN安装了`Term::ReadKey`和`Term::ReadLine`模块，就能得到类似于GNU `readline(3)`提供的所有编辑功能。



如果你的系统上安装了`emacs`，它可以与Perl调试器交互（类似于与C调试器的交互），来提供一个集成软件开发环境。Perl提供了一个启动文件，使`emacs`相当于一个语法导向的编辑器，能够理解Perl的（部分）语法。查看Perl源代码版本的`emacs`目录。`vi`用户还应该看看`vim`（以及用得很多的版本`gvim`），这些编辑器会用不同颜色显示Perl关键字。

我们的一位同事（Tom）也做了一个类似的工作，可以支持与任何开发商提供的`vi`和X11窗口系统的交互。这个工作类似于`emacs`提供的集成多窗口支持，由调试器驱动编辑器。不过，写这本书时，它在Perl发行版中的最终定位还不确定。但我们认为你应该知道存在这种可能。

## 用初始文件定制

可以建立包含初始代码的`.perl5db`或`perl5db.ini`文件（取决于你的操作系统）来完成一些定制。这个初始文件包含Perl代码，而不是调试器命令，而且要在查看`PERL5DB_OPTS`环境变量之前处理。例如，可以向`%DB::alias`散列增加记录来建立别名，如下所示：

```
$alias{len} = 's/^len(.*)/p length($1)/';
$alias{stop} = 's/^stop (at|in)/b/';
$alias{ps} = 's/^ps\b/p scalar /';
$alias{quit} = 's/^quit(\s*)/exit/';
$alias{help} = 's/^help\s*$|/h/';
```

可以在你的初始文件中使用函数调用调试器的内部API来改变选项：

```
parse_options("NonStop=1 LineInfo=db.out AutoTrace=1 frame=2");
```

如果你的初始文件定义了子例程`afterinit`，这个函数会在调试器初始化结束之后调用。初始文件可能位于当前目录，也可能在主目录。由于这个文件可能包含任意的Perl命令，出于安全性的考虑，必须为超级用户或当前用户所有，而且除了它的所有者之外不允许其他任何人写这个文件。

如果你想修改调试器，可以将Perl库的`perl5db.pl`复制为另一个名字，并适当修改来满足你的要求。然后可以设置`PERL5DB`环境变量，如下所示：

```
BEGIN { require "myperl5db.pl" }
```

最后一点，还可以使用`PERL5DB`通过直接设置内部变量或者调用内部调试器函数来定制调试器。不过要当心，这里没有明确说明或者在线`perldebug`、`perldebbugs`或DB手册页上都没有介绍的变量和函数都认为只在内部使用，这些变量和函数可能会修改而不做任何通知。

## 调试器选项

调试器有很多选项，可以用`O`命令来设置，为此可以从环境交互式设置，也可以通过初始文件来设置。

### `recallCommand`, `ShellBang`

用于重新调用一个命令或创建一个shell的字符。这两个选项都默认设置为`!`。

### `pager`

这个程序应用于分页管道命令（以`|`字符开头的命令）的输出。默认将使用`$ENV{PAGER}`。由于调试器会用当前终端特性来设置粗体和加下划线，如果选择的分页程序不能不加改变地传递转义序列，一些调试器命令的输出通过分页程序发送时可能不可读。

### `tkRunning`

在Tk模块下运行，同时提示输入（用`ReadLine`）。

### `signalLevel`, `warnLevel`, `dieLevel`

设置详细程度。默认情况下调试器不处理异常和警告，因为提示异常和警告可能会中断本来正确运行的程序。

要禁用这个默认的安全模式，可以把这些值设置为大于0的某个级别。级别为1时，接收到某种警告（这通常很讨厌）或异常（这通常很有意义）时可以得到回溯记录。遗憾的是，调试器无法区分致命异常和非致命异常。如果`dieLevel`为1，那么非致命异常也会记录，而且如果它们来自`eval`计算的字符串或者来自你想要加载的模块中的某个`eval`，会被随意修改。如果`dieLevel`为2，调试器不关心警告和异常来自哪里：调试器会抢占异常处理器，并打印一个跟踪记录，然后用它自己的方式修改所有异常。这对于跟踪可能很有用，不过往往会让那些强调异常处理的程序很麻烦。

未捕获的`INT`、`BUS`或`SEGV`信号到达时，调试器会尝试打印一个消息。如果你正处在一个很慢的系统调用中（如`wait`或`accept`，或者从键盘或套接字读（`read`）），而且没有建立你自己的`$SIG{INT}`处理器，就不能通过`Control-C`回到调试器，因为调试器自己的`$SIG{INT}`处理器并不知道要生成一个异常用`longjmp(3)`退出这些很慢的系统调用。

### `AutoTrace`

设置跟踪模式（类似于`t`命令，不过可以放在`PERLDB_OPTS`中）。

### `LineInfo`

指定将行数信息打印到哪个文件或管道。如果这是一个管道（比如`|visual_perldb`），则使用一个短消息。这正是与从编辑器或可视化调试器交互时所使用的机制，如特殊的`vi`或`emacs`钩子，或者`ddd`图形化调试器。

**inhibit\_exit**

如果为0，允许跳出到程序末尾。

**PrintRet**

设置r命令后（默认），打印返回值。

**ornaments**

影响命令行的屏幕显示（参见Term::ReadLine的在线文档）。目前没有办法禁用ornaments，对于某些显示器或者某些分页程序，可能会显示一些不可识别的输出。我们认为这是一个bug。

**frame**

影响进入和退出子例程时消息的打印。如果frame & 2为false，只是进入子例程时打印消息（如果与其他消息混杂在一起，在退出子例程时打印消息可能很有用）。

如果frame & 4为true，会打印函数的参数以及上下文和调用者信息。如果frame & 8为true，会对打印的参数启用重载字符串化和已绑定（tie）FETCH。如果frame & 16为true，会打印子例程的返回值。

参数表的截断长度由下一个选项控制。

**maxTraceLen**

frame选项的第4位设置为1时，这个选项设置参数表的截断长度。

下面的选项会影响V、X和x命令的作用：

**arrayDepth, hashDepth**

只打印前n个元素。如果忽略n，则打印所有元素。

**compactDump, veryCompact**

改变数组和散列输出的风格。如果启用compactDump，短数组可能打印在一行上。

**globPrint**

打印类型团的内容。

**DumpDBFiles**

显示包含所调试文件的数组。

**DumpPackages**

显示包的符号表。

**DumpReused**

显示“重用”地址的内容。



quote, HighBit, undefPrint

改变字符串显示的风格。quote的默认值是auto，可以将它设置为"或'，分别启用双引号格式或单引号格式。默认地，设置了高位的字符会逐字打印。

UsageOnly

启用这个选项时，并不是显示包的变量的内容，调试器会根据包变量中找到的字符串的总大小得到每个包的基本内存使用转储。由于使用了包符号表，会忽略词法作用域变量。

## 不被注意的执行

在启动期间，会从\$ENV{PERLDB\_OPTS}初始化选项。可以在这里放置初始化选项TTY、noTTY、ReadLine和and NonStop。

如果你的初始文件包含：

```
parse_options("NonStop=1 LineInfo=tperl.out AutoTrace");
```

那么程序可以不加人工干预地运行，并将跟踪信息放在文件db.out中（如果中断了程序，要想看到跟踪信息，最好将LineInfo重置为/dev/tty）。

下面的选项只能在启动时指定。要在初始文件中设置这些选项，可以调用parse\_options("OPT=VAL")。

TTY

用于调试I/O的终端。

noTTY

如果设置了这个选项，调试器会进入NonStop模式，而不会连接到一个终端。如果被中断（或者从Perl程序显式地设置\$DB::signal或\$DB::single而进入调试器），调试器会连接到启动时在TTY选项中指定的一个终端，或者连接到你选择的Term::Rendezvous模块在运行时找到的一个终端。

这个模块要实现一个名为new的方法，它要返回一个包含两个方法的对象：IN和OUT。这些方法应当为调试器返回用作为输入和输出的文件句柄。new方法需要一个参数，其中包含启动时的\$ENV{PERLDB\_NOTTY}值，或者是/tmp/perldebugtty\$\$。调试器不会检查这个文件的所有权以及是否可写，所以理论上讲存在着安全隐患。

ReadLine

如果为false，会禁用调试器中的ReadLine支持，以便调试本身使用了ReadLine模块的应用。

## NonStop

如果设置了这个选项，调试器会进入非交互模式，直到被中断，或者直到你的程序设置了`$DB::signal`或`$DB::single`。

有时选项可以缩写为首字母，不过我们建议你还是将它完整地拼出，这样既便于阅读，也有利于将来的兼容性。

下面给出一个使用`PERLDB_OPTS`环境变量自动设置选项的例子<sup>注1</sup>。它会以非交互的方式运行你的程序，每次进入一个子例程时会打印信息，另外还会对所执行的每一行打印信息。调试器的跟踪输出将放在`tperl.out`文件中。这允许你的程序仍使用常规的标准输入和输出，不用担心跟踪信息混杂其中。

```
$ PERLDB_OPTS="NonStop frame=1 AutoTrace LineInfo=tperl.out" perl -d myprog
```

如果中断这个程序，需要迅速重置为`0` `LineInfo=/dev/tty`或你的平台上任何有意义的设置，否则你不会看到调试器的信息输出。

## 调试器支持

Perl为编译时和运行时都提供了特殊的调试钩子来创建调试环境（如标准调试器）。不要把这些钩子与`perl -D`选项混淆，后者仅在Perl构建包含`-DDEBUGGING`支持时才能使用。

例如，如果从包`DB`调用Perl的内置`caller`函数，调用相应栈帧时提供的参数会复制到`@DB::args`数组。调用Perl时如果提供了`-d`开关，会启用以下额外的特性：

- Perl把`$ENV{PERL5DB}`的内容（或者如果没有`$ENV{PERL5DB}`，则将`BEGIN {require 'perl5db.pl'}`的内容）插入到程序的第一行前面。
- 数组`@{"_<$filename"}`包含`$filename`行，对应由Perl编译的所有文件，以及包含子例程或者当前正在执行的用`eval`计算的字符串。对于用`eval`计算的字符串，相应的`$filename`看起来形如（`eval 34`）。正则表达式中的代码断言形如（`re_eval 19`）。
- 散列`%{"_<$filename"}`包含以行号为键的断点和动作。可以设置单个散列记录而不是整个散列。Perl只关心这里的真值，尽管`perl5db.pl`使用的值形如`"$break_condition\0$action"`。这个散列中的值在上下文中是有魔法的：如果行不可中断则为`0`。
- 包含子例程或者当前正在执行的计算字符串也同样如此。对于用`eval`计算的字符串，相应的`$filename`形如（`eval 34`）或（`re_eval 19`）。
- 标量`${"_<$filename"}`包含`"_<$filename"`。包含子例程或者当前正在执行的计算字

---

注1： 我们使用`sh` shell语法来显示环境变量设置。如果使用其他shell要相应进行调整。

字符串也同样如此。对于用eval计算的字符串，相应的\$filename形如 (eval 34) 或 (re\_eval 19)。

- 各个require加载的文件完成编译之后且在其执行之前，如果子例程DB::postponed存在，会调用DB::postponed(\*{"\_<\$filename"})。这里\$filename是require加载的文件的扩展名字（如%INC中的值）。
- 编译各个子例程subname之后，检查\$DB::postponed{subname}是否存在。如果这个键存在，而且如果子例程DB::postponed也存在，会调用DB::postponed(subname)。
- 维护散列%DB::sub，其键为子例程名，值形式为filename:startline-endline。对于eval中定义的子例程，filename形式为 (eval 34)，或者对于正则表达式代码断言中的子例程，形式为 (re\_eval 19)。
- 如果程序执行到某个可能包含断点的位置，如果变量\$DB::trace、\$DB::single或\$DB::signal中任意一个为true，会调用DB::DB子例程。这些变量不可置为local变量。在DB::DB内部执行时（包括从它调用的函数），会禁用这个特性，除非\$^D & (1<<30)为true。
- 程序执行到一个子例程调用时，会取代为一个&DB::sub(args)调用，\$DB::sub将包含所调用子例程的名字。如果子例程在DB包中编译，则并非如此。

注意如果&DB::sub需要外部数据来完成工作，在它完成工作之前不能做任何子例程调用。对于标准调试器，\$DB::deep变量（达到一个强制中断之前调试器中允许的最大递归深度）就是这样一个例子，体现了这种依赖性。

## 编写自己的调试器

最小的可用调试器只有一行：

```
sub DB::DB {}
```

因为它什么也做不了，可以很容易地通过PERL5DB环境变量来定义：

```
$ PERL5DB="sub DB::DB {}" perl -d your-program
```

还可以如下创建一个同样很小但稍有用处的调试器：

```
sub DB::DB {print ++$i; scalar <STDIN>}
```

这个小调试器会打印遇到的每一个语句的序号，它会等待你按下回车之后再继续。

下面这个调试器尽管看上去很短小，不过功能已经很强了：

```
{  
    package DB;  
    sub DB {}  
}
```



```

        sub sub {print ++$i, " $sub\n"; &$sub}
    }

```

它会打印子例程调用的序号，以及所调用子例程的名字。需要说明，就像我们这里所做的，必须从包DB编译`&DB::sub`。

如果基于当前调试器建立你的新调试器，可以利用一些钩子帮你完成定制。启动时，调试器从当前目录或你的主目录读取初始文件。读取这个文件之后，调试器读取`PERLDB_OPTS`环境变量，并将它作为0 ...行的剩余部分进行解析（就像你在调试器提示窗口中输入的一样）。

调试器还维护一些魔法内部变量，如`@DB::dbline`和`%DB::dbline`，它们分别是`@{ "::_<current_file" }`和`%{ "::_<current_file" }`的别名。这里`current_file`是当前选择的文件，可能用调试器的`f`命令显式选择，也可能从执行流隐式选择。

可以利用一些函数帮助完成定制。`DB::parse_options(STRING)`会像0选项一样解析一行。`DB::dump_trace(SKIP[, COUNT])`跳过指定数目的帧，并返回一个列表，其中包含有关调用帧的信息（如果没有指定`COUNT`，则是所有帧）。列表中每一项都是一个散列引用，包含键“`context`”（`.`，`$`或`@`）、“`sub`”（子例程名或有关`eval`的信息）、“`args`”（`undef`或一个数组引用）、“`file`”和“`line`”。`DB::print_trace(FH,SKIP[, COUNT[, SHORT]])`将有关调用者帧的格式化信息打印到指定的文件句柄。最后两个函数可以很方便地作为调试器`<`和`<<`命令的参数。

你不需要了解所有这些内容，其实我们大多数人都并不了解。实际上，需要调试一个程序时，通常只需要在程序中插入一些`print`语句，然后重新运行程序。

原先我们甚至要记住先打开警告。这通常会立即发现问题，避免以后为解决这些问题而满头大汗。不过，如果这样不行，最好知道在那个`-d`开关后面还有一个可爱的调试器在等着你，它能为你做几乎任何事情，除了不能帮你找你身上的臭虫。

不过，关于定制调试器如果你只想记住一点，这最重要的一点可能是：不要只认为让Perl不满意的东西才是bug，如果你的程序让你不满意，它也是一个bug。之前，我们给出了两个相当简单的定制调试器。在下一节中，我们会展示一个不同类型的定制调试器例子，这个例子可能会帮助你调试那些称为“到底完成没有？”的bug（也可能没有帮助）。

## Perl性能测试

写这本书时，Perl提供了一个名为`Devel::DProf`的性能测试工具。不过，等你读到这本书时，这个模块可能已经没有了。Perl v5.16计划与这本书上架同期发布，它删除了这个老的性能测试工具。使用性能测试工具的大多数人都转向另一个性能测试工具

Devel::NYTProf。我们仍将为你介绍Devel::DProf，因为它还在Perl中，不过我们还会介绍那个新模块，尽管目前Perl还没有提供那个模块。

这两个性能测试工具都不是轻量级的，它们也不是你的唯一选择。CPAN还包含一个Devel::SmallProf，它能报告执行每行程序花费的时间。如果你在使用某个特定的Perl构造，开销过大，这个工具可以帮你找出问题。大多数内置函数都很高效，不过很容易无意地写一个有问题的正则表达式，其开销相对于输入大小呈指数增长。参见第21章“效率”一节，从中可以得到更多有帮助的提示。

## Devel::DProf

你想让你的程序更快吗？嗯，当然了。不过首先你要停下来问问自己，“我真的需要花时间让这个程序更快吗？”<sup>注2</sup> 消遣性的优化可能很有趣<sup>注2</sup>，不过通常你还可以更好地利用你的时间。有时，你只需要提前规划，然后在你喝咖啡的时候开始运行程序（或者拿它作为休息的借口）。不过，如果你的程序必须更快地运行，就应该对它进行性能测试。性能测试工具可以告诉你程序中的哪一部分执行的时间最长，这样你就不用浪费时间去优化那些对整体执行时间影响不大的子例程。

Perl提供了一个性能测试工具：Devel::DProf模块。输入以下命令，可以用它对mycode.pl中的Perl程序完成性能测试：

```
% perl -d:DProf mycode.pl
```

尽管我们把它称为性能测试工具，因为这确实是它的工作，但DProf采用的机制与本章前面讨论的内容完全相同。DProf只是一个调试器，它会记录Perl进入和离开每个子例程的时间。

性能测试脚本终止时，DProf把这些时间信息转储到一个名为tmon.out的文件。Perl提供的dprofpp程序知道如何分析tmon.out并生成一个报告。还可以使用dprofpp并结合-p开关（稍后介绍）作为整个过程的前端。

给定以下程序：

```
outer();

sub outer {
    for (my $i=0; $i< 100; $i++) { inner() }
}

sub inner {
    my $total = 0;
```

---

注2： Nathan Torkington这么说，本书这一部分就出自他之手。



```

    for (my $i=0; $i< 1000; $i++) { $total += $i }
}

inner();

```

*dprofpp*的输出为:

```

Total Elapsed Time = 0.537654 Seconds
  User+System Time = 0.317552 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c Name
 85.0  0.270   0.269   101   0.0027 0.0027 main::inner
 2.83  0.009   0.279     1   0.0094 0.2788 main::outer

```

注意, 百分数累加的总和没有达到100。实际上, 在这里差距还很大, 由此可以提示你需要让程序运行更长时间。一般经验是, 收集的性能测试数据越多, 你的统计样本就越好。如果将外循环增加到运行1000次而不是100次, 可以得到更精确的结果:

```

Total Elapsed Time = 2.875946 Seconds
  User+System Time = 2.855946 Seconds
Exclusive Times
%Time ExclSec CumulS #Calls sec/call Csec/c Name
 99.3   2.838   2.834   1001 0.0028 0.0028 main::inner
 0.14   0.004   2.828     1 0.0040 2.8280 main::outer

```

第一行报告了程序从开始到结束运行所用的时间。第二行显示了两个数字之和: 执行代码花费的时间即用户时间 (“user”) 和操作系统执行你的代码的系统调用所花费的时间即系统时间 (“system”)。我们必须原谅这些数字有些不太精确, 计算机的时钟并不是百万分之一秒滴答一次。如果幸运, 它可能百分之一秒滴答一次。

可以利用*dprofpp*的命令行选项改变 “user+system” 时间。-r显示耗用时间, -s只显示系统时间, -u只显示用户时间。

报告的其余部分是各子例程所用时间的一个明细。“Exclusive Times” (独占时间) 行表示子例程outer调用子例程inner时, inner花费的时间不会统计到outer的时间中。要改变这一点, 希望将inner的时间统计到outer的时间中, 可以为*dprofpp*提供-I选项。

对于每个子例程, 会报告以下内容: %Time, 这个子例程调用所花费时间的百分比; ExclSec, 这个子例程花费的时间 (单位为秒), 但不包括从它调用的其他子例程花费的时间; CumulS, 这个子例程以及从它调用的子例程花费的总时间 (即包含时间, 单位为秒); #Calls, 调用这个子例程的次数; sec/call, 调用这个子例程的平均时间 (单位为秒), 不包括从它调用的子例程花费的时间; Csec/c, 调用这个子例程以及从它调用的其他子例程的平均时间 (单位为秒)。

其中, 最有用的数是%Time, 它会告诉你时间都花到什么地方了。在这里, 最花时间的inner子例程, 所以我们想优化这个子例程, 或者想找出一种算法能减少这个子例程



的调用。*dprofpp*的选项还允许访问其他信息，或者可以改变计算时间的方式。还可以让*dprofpp*先运行脚本，这样你就不必记着使用*-d:DProf*开关：

***-p SCRIPT***

告诉*dprofpp*要对给定的*SCRIPT*完成性能测试，然后解释其性能测试数据。参见*-Q*。

***-Q*** 结合*-p*使用，告诉*dprofpp*完成脚本的性能测试之后退出，而不解释数据。

***-a*** 按子例程名以字母顺序对输出排序，而不是按时间百分比递减的顺序排序。

***-R*** 独立统计同一个包中定义的匿名子例程。默认行为是将所有匿名子例程统计为一个子例程，名为`main::_ _ANON_ _`。

***-I*** 显示所有子例程时间（包括所调用的下一级子例程时间）。

***-l*** 按子例程调用数排序。这个选项有助于找出可以内联的子例程。

***-O COUNT***

只显示最上面*COUNT*个子例程。默认为15。

***-q*** 不显示列标题。

***-T*** 将子例程调用树显示到标准输出。不显示子例程统计信息。

***-t*** 将子例程调用树显示到标准输出。不显示子例程统计信息。同一个调用级别上多次（连续）调用的函数只显示一次，并显示重复次数。

***-S*** 按照子例程相互调用的方式生成输出：

```
main::inner x 1      0.008s
main::outer x 1      0.467s = (0.000 + 0.468)s
main::inner x 100    0.468s
```

可以这样来理解：顶级程序调用一次*inner*，它运行了0.008s（耗用时间）；顶级程序调用一次*outer*，运行时间0.467s（包含*outer*本身的0s，另外从*outer*调用的子例程运行了0.468s），顶级程序调用*inner*100次（运行时间0.468s）。明白了吗？

同一级分支（例如，调用一次的*inner*和调用一次的*outer*）按其包含时间排序。

***-U*** 不排序。按原性能测试中的顺序显示。

***-v*** 按各调用期间子例程花费的平均时间排序。这有助于找出可以通过内联子例程体进行优化的子例程。

***-g subroutine***

忽略子例程，但*subroutine*以及由它调用的子例程除外。

其他选项见*dprofpp(1)*（标准手册页）的描述。

## Devel::NYTProf

Devel::NYTProf模块是发源于*New York Times*，由Adam Kaplan编写，不过目前在*Times*以外维护。它的速度很快（用C编写），功能很强大，而且可以得到很漂亮的报表。这是目前可用的最快的语句和子例程性能测试工具，我们没有足够的篇幅让你全面了解它的所有优点。你可以从CPAN下载，然后像使用其他调试器一样使用：

```
% perl -d:NYTProf your_program
```

完成性能测试后，可以查看结果HTML文件。第一个HTML文件（见图18-1）是一个总结：

```
% nytprofhtml --open
```

可以在NYTPROF环境变量中设置多个选项。例如，可以告诉性能测试工具何时开始：立即开始、INIT阶段开始还是进入END时再开始：

```
% env NYTPROF=start=init perl -d:NYTProf your_program
```

有关的更多详细内容参见模块文档。现在来杯咖啡稍做休息。下一节你会需要它。

# Performance Profile Index

For tools/pod2docbook

Run on Sun Oct 23 21:05:53 2011  
Reported on Sun Oct 23 21:09:25 2011

Profile of tools/pod2docbook for 27.9s (of 38.2s), executing 9289551 statements and 3270739 subroutine calls in 74 source files and 17 string evals.

Jump to file... :

## Top 15 Subroutines

Calls	P	F	Exclusive Time	Inclusive Time	Subroutine
539364	9	1	2.14s	2.14s	Local::DocBook::get_pad
127890	21	1	1.75s	4.09s	Local::DocBook::emit
2982	1	1	1.52s	25.2s	Pod::Simple::BlackBox::parse_lines
49937	1	1	1.17s	5.49s	Local::DocBook::handle_text
9532	1	1	1.17s	2.08s	Pod::Simple::BlackBox::treelet_from_formatting_codes
36133	2	1	1.17s	1.78s	Local::DocBook::make_curly_quotes
154912	16	1	1.11s	1.72s	Local::DocBook::add_to_pad
327281	13	2	1.05s	1.05s	Pod::Simple::BlackBox::CORE:match (opcode)
136126	15	1	1.01s	1.53s	Local::DocBook::clear_pad
156	6	1	952ms	952ms	main::CORE:subst (opcode)
93529	73	1	786ms	4.84s	Local::DocBook::add_xml_tag
34251	2	2	774ms	15.8s	Pod::Simple::BlackBox::traverse_treelet_bit (recurses: max depth 3, inclus
59366	1	1	680ms	1.95s	Encode::decode
26	1	1	631ms	26.0s	Pod::Simple::parse_file
442198	29	1	619ms	671ms	Local::DocBook::CORE:subst (opcode)

See all 1230 subroutines

You can view a treemap of subroutine exclusive time, grouped by package.  
NYTProf also generates call-graph files in Graphviz format: inter-package calls, all inter-subroutine calls (probably too complex to render easily).



CPAN最初只是一个Perl软件库，逐步演化成为基于这个软件库构建的一个松散的服务集合。人们谈到“CPAN”时，可能是指与它有关的任何方面，因为人们总把与这个核心软件库关联的东西混杂在一起。

## 历史

1993年末，Tim Bunce、Jarkko Hietaniemi和Andreas König建立了一个perl爱好者邮件列表，共同讨论一个想法，希望为互联网上有关Perl 4的所有东西建立一个存档。那一年Perl 5已经开始开发，其中一个主要特性就是要提供一个可扩展的模块系统，允许人们扩展这个语言而无需改变perl。Jared Rhine提出了中心存储库的建议，不过反响不大。他的想法来自于CTAN，即TeX综合典藏网（Comprehensive TeX Archive Network）。

几年之后，Jarkko重新启动了这个想法，并在`ftp://ftp.cpan.org`建立了一个FTP存档。稍后，Andreas König建立了PAUSE（Perl作者上传服务器，Perl Authors Upload Server），使人们可以通过这个途径对核心存储库做出贡献。大多数人所认为的“CPAN”部分（即模块部分）实际上只是CPAN从PAUSE镜像的两个目录。不过，CPAN远不只是这些。

另外还有一些镜像主CPAN网站的服务，它们允许在全球范围内快速而方便地访问这个CPAN。现在六大洲约有300个公共镜像。任何人都能镜像CPAN的所有内容，来创建一个新的公共镜像，或者甚至可以创建一个私人镜像供自己使用<sup>注1</sup>。

随着CPAN变得越来越流行，围绕它又开发了其他一些项目。Graham Barr在`http://search.`

---

注1： 参见“`How to mirror CPAN`”（如何镜像CPAN），网址是：`http://www.cpan.org/misc/how-to-mirror.html`。

*cpan.org*增加了一个搜索界面。Barbie提出了CPAN Testers的想法，即测试CPAN上的每一个模块。David Cantrell开发了CPANdeps，可以合并模块及其所有依赖包的测试结果。Moritz Onken创建了一个第二代搜索和聚合网站，将MetaCPAN作为“Google Summer of Code”项目的一部分。围绕真正的CPAN（即前面所说的核心存储库）还建立了很多其他服务。

## 存储库之旅

CPAN上的大多数文件都来自PAUSE，PAUSE提供了*authors*和*modules*目录。不过除了模块外，CPAN还有很多内容。下面对其中比较有意思的部分做一个简要介绍。

### *authors*

这是从PAUSE镜像的一个目录，包含很多子目录，在id子目录下按贡献者的作者ID来组织。目录的第一层是作者ID的首字母，第二层是前两个字母，最后一层是完整的作者ID。例如，对于作者NANIS（Sinan Ünür），*authors*下的路径就是id/N/NA/NANIS。这个目录下是Sinan上传（而且没有删除）的所有内容，参见BackPAN。

有些作者会有一个以其全名命名的目录，如Hugo\_van\_der\_Sanden。如果没有那么多作者，*authors*目录本来采用的是一个更扁平的结构。随着CPAN越来越流行，现在已超过9000个注册作者，PAUSE不得不将作者名划分为3层结构。

### *doc*

这个目录用来存放Perl文档，以及有关的各种评论，不过现在已经不再维护这个目录。其中还有一些有意思的内容，但它不再是Perl信息的主要来源。要得到在线文档，可以使用<http://perldoc.perl.org>得到核心Perl信息，或查看某个Perl模块网站来得到相关的模块文档。

### *modules*

奇怪的是，这个目录中并不是你要找的模块，而是一些特殊的索引文件，CPAN客户程序使用这些索引文件将包名（如Mojolicious）转换为它在*authors*下的具体路径，对于包Mojolicious，相应的路径就是*authors/id/S/SR/SRI/Mojolicious-1.99.tar.gz*（或者Mojolicious最新版本所在的任何位置）。

这里还有一些子目录，如*by-module*，它会按模块名而不是按作者组织模块。其中包含的是一些符号链接，指向实际文件所在的*authors*目录。

另外还有一个*by-category*目录，如今这个目录已经没有太大用处。CPAN拥有如此之多的模块之前，CPAN管理者曾经希望对各个模块分类，以便你按类别<sup>注2</sup>导航来找出你想要的模块。与在CPAN的层层组织中漫游相比，直接搜索变得更为流行，所以

---

注2： 还有人记得搜索盛行之前的Yahoo!吗？



“模块列表”日渐荒废，很快就过时了。这个目录依赖于模块作者向PAUSE注册模块并进行分类，不过如今没有多少人这么做了。

### *ports*

这个目录包含源代码，有时还包含Perl移植版本的预编译可执行映像，这主要针对标准版中不直接支持的那些操作系统，或者是极难得到编译器的那些Perl版本。这些移植版本完全由其各自的作者努力而成，可能与本书中介绍的不完全一样。如今，很少有系统需要特殊的移植版本。不过不管怎样，这个目录的索引文档很有意思，很值得看一看，因为其中可能详细说明了Perl何时成为各个系统开发商标准安装的一部分。

### *scripts*

这个目录包含不同种类的Perl程序，数量不多，主要是人们发布独立程序的那个时期遗留下来的。作者们将其程序放在*scripts*中，并在程序文档中包含特殊的pod标题。不过，几乎没有人再这么做了。现在人们总是将程序作为普通Perl模块的一部分上传，大多上传到App::namespace中。与其说CPAN有不错的脚本友好性，不如说它有很好的模块友好性。

*src* 可以在这个目录中找到标准Perl发布版本的源代码。实际上，这里有两个标准Perl发布版本，一个标为*maint*，另一个标为*devel*。Perl有两个开发路线，在实际工作中应当使用*maint*。而*devel*是试验性的，Perl开发人员可以在这里尝试新特性、新代码，以及其他新鲜事物，这些新特性可能存在问题，还不能稳定使用。

要了解究竟是哪种版本，可以查看版本号，如5.14.2。第一个数字是主版本号，表示这是Perl v5。第二个数是次版本号<sup>注3</sup>。如果这个次版本号是偶数，则是一个维护版本。所以版本5.10.1、5.12.4和5.14.2都是稳定的版本，因为10、12和14都是偶数。如果版本号是奇数，如5.15.3，这就是一个试验版本，因为15是奇数。

这个目录中有两个链接，它们总是得到最新的版本，而不论实际版本是什么。*latest.tar.gz*和*maint.tar.gz*指向最新的维护版本<sup>注4</sup>。CPAN维护人员不建议使用这些术语，因为有些人不知道它们表示什么。

## 创建一个MiniCPAN

你可以自己镜像CPAN，不过写这本书时，如果你要自己镜像，就必须同步24000个模块版本，总共占13 GB磁盘空间。由于PAUSE只对最新的模块版本建立索引，你可能并不需要

---

注3：这不是说范围小。这也是大版本的号。可以这样来考虑，把“Perl 5”认为是语言，下一个号是主版本。

注4：Perl开发人员正式支持最后两个维护版本。如果当前版本是Perl v5.16，这意味着v5.14也得到正式支持，不过不会正式支持v5.12。有关的详细信息参见*perlpolicy*。



其中大部分模块。大多数情况下，可能只需要安装最新的版本。正因如此，2002年Randal Schwartz创建了`minicpan`，并在`Linux Magazine`中做了介绍<sup>注5</sup>。他把本地CPAN的占用空间缩减了80%，brian d foy还为此发明了一个新词“Schwartz因子”来度量缩减程度。

CPAN::Mini包含你需要的很多工具。这个模块并不是标准库的一部分，所以你必须自己安装（参见本章后面的介绍）。

首先，建立配置，指出你希望从哪里获取新数据，以及希望将数据存储在哪里：

```
local: /Users/Amelia/MINICPAN
remote: http://cpan.example.com/
```

运行`minicpan`来创建这个“纤细”的存储库：

```
% minicpan
Using config from /Users/Amelia/.minicpanrc
Updating /Users/Amelia/MINICPAN
Mirroring from http://cpan.example.com/
=====
authors/01mailrc.txt.gz ... updated
modules/02packages.details.txt.gz ... updated
modules/03modlist.data.gz ... updated
authors/id/A/AA/AAR/Math-Clipper-1.01.tar.gz ... updated
authors/id/A/AA/AAR/CHECKSUMS ... updated
```

将CPAN客户程序指向这个存储库，现在不论什么场合你都能安装模块，即使你在火车上、飞机上、汽车上，或者身在黑岩沙漠中，甚至停电都不会影响你。嗯，当然你的电池可不能没电。

你可以精细地控制要同步哪些模块，尽管为此需要编写你自己的`minicpan`程序来使用这些控制工具。`module_filters`和`path_filters`允许你使用正则表达式或子例程引用指示要跳过哪些模块或作者。如果某个模式匹配，或者子例程返回true，这会让CPAN::Mini跳过相应的模块：

```
use CPAN::Mini;

CPAN::Mini->update_mirror(
    remote      => "http://cpan.mirrors.comintern.su",
    local       => "/usr/share/mirrors/cpan",
    force       => 0,
    module_filters => [ qr/Acme/i ],
    path_filters  => [
        qr/RJBS/,
        sub { $_[0] =~ /SUNGO/ }
    ],
);
```

---

注5： 参见<http://www.stonehenge.com/merlyn/LinuxMag/col42.html>。

# CPAN生态系统

CPAN实际上是一个服务网络，在上传模块的人到安装模块的人之间有很多步骤。这一章并不会介绍每一个细节，这里只强调最主要的部分。

## PAUSE

PAUSE (<http://pause.perl.org>) 是贡献者进入CPAN的大门。在上传任何模块之前，你首先需要有一个帐户。申请帐户是免费的，而且很容易。PAUSE的某个管理者会检查你的应用，主要检查它不是机器人自动上传的程序，然后建立帐户。

一旦有了帐户，就可以把你的作品上传到PAUSE了。几乎可以上传你喜欢的任何东西。PAUSE并不关心你做了什么或者你做得怎么样。

上传一个模块时，PAUSE索引人员会查看你的存档，查找你可能使用的Perl命名空间。对于你能使用哪些命名空间并没有任何限制，不过PAUSE维护了一个列表，其中包含它认为有权修改命名空间的人。

- 第一个使用某个命名空间的作者会得到*first-come*权限，并成为主要维护者（primary maintainer）。
- 主要维护者（primary maintainer）可以为另一个作者（或很多作者）分配合作维护者（co-maintainer）权限。
- 主要维护者可以放弃其权限，转交给另一个作者。

如果你上传了一个模块，它使用了一个命名空间，而你对这个命名空间没有上述权限，PAUSE索引人员就会拒绝对这个模块建立索引，并向你发出一个错误。不过它会接受你的上传，你上传的模块仍会出现在CPAN上。人们可以下载这个模块，然而，由于索引人员没有对它建立索引，所以你的模块不会出现在PAUSE创建的数据库中。既然你的模块没有出现在数据库中，CPAN客户程序就无从知道它的存在，当然也不会安装它。PAUSE会跳过你的模块，整个世界都将它冷落在一边。

## 搜索CPAN

有两个主要的CPAN搜索网站，它们的功能类似。这两个网站汇总了其他CPAN项目的特定模块版本链接：

- CPAN Search (<http://search.cpan.org>)
- MetaCPAN (<https://www.metacpan.org>)

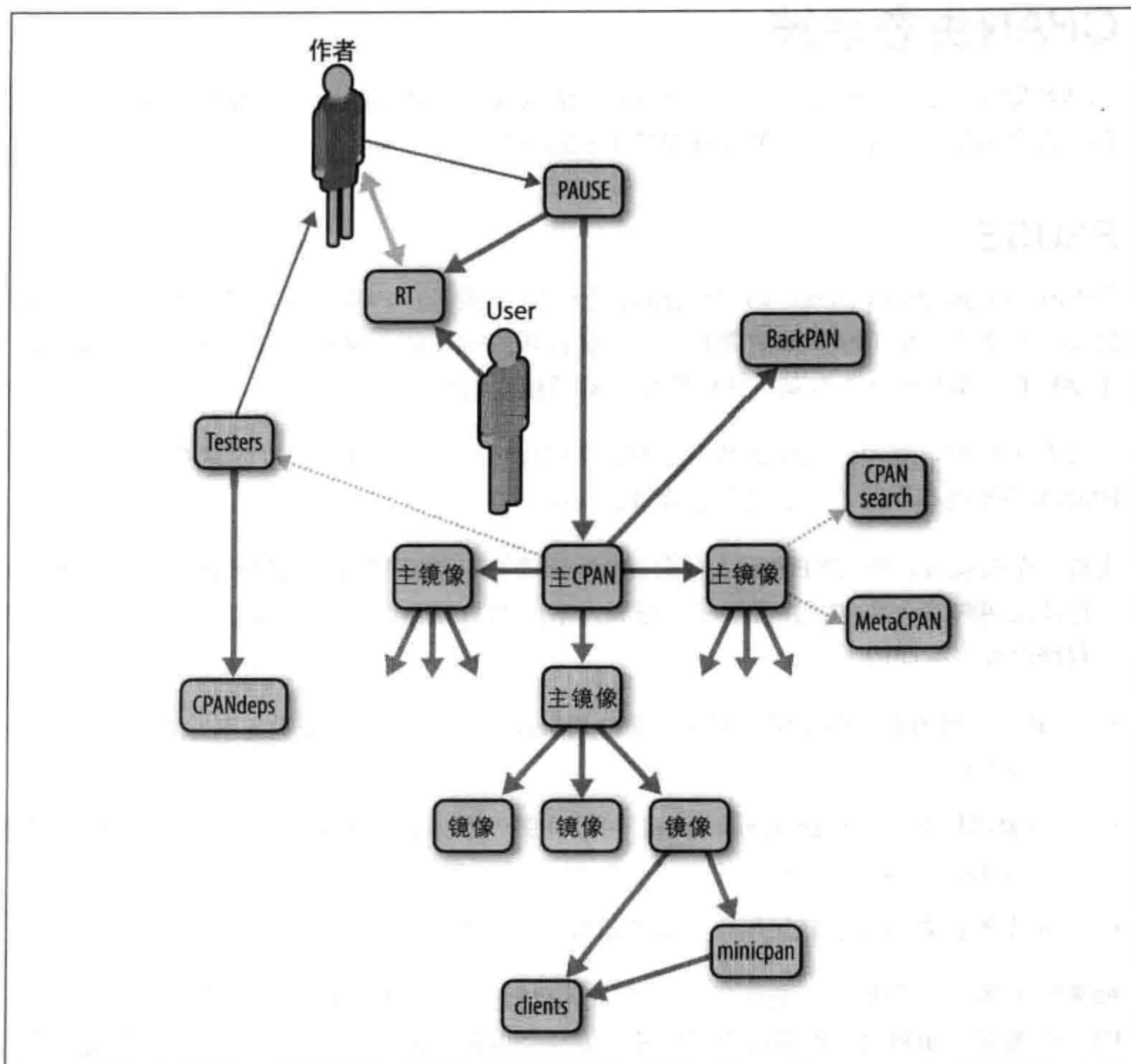


图19-1：CPAN生态系统图

## 测试

Perl有一种很棒的测试文化。一旦有人向PAUSE上传一个新的模块版本，就会有一个由不同规格和大小的测试机器组成的松散联盟来下载、构建和运行其测试套件，这个联盟称为CPAN Testers (<http://testers.cpan.org>)。这个组织着力在所能找到的尽可能多的平台和Perl版本上测试尽可能多的模块。

通过这种方式，孤独的Perl作者可以在某一个体系结构上完成开发，通过简单的上传，就能得到其他体系结构以及跨多个Perl版本的结果。而且所有这些都是免费的！作者们通过阅读CPAN Testers wiki (<http://wiki.cpantesters.org>)可以得到详细的说明，以保证他们的模块做到“对测试者友好”。



这也适用于CPAN模块的用户。人们可以查看测试报告，了解一个特定的模块进展如何。David Cantrell的CPANdeps (<http://deps.cpan testers.org>) 通过平台和Perl版本的一个矩阵来提供测试报告，另外还对所有模块依赖性（安装“成功概率”）的测试报告提供了一个总结。

## Bug跟踪

由于Perl和CPAN不是单一的集中式项目，所以没有一个地方来报告或了解bug。尽管很多人会在私人email中直接报告bug，但是这样并不能创建一个公共记录，无法让所有人都了解、评论以及（有可能）修正。只有能够公开访问，开源才有意义，如果只是把消息放在某个人的email中，这个消息并没有向世界公开供其他人检查审阅。

### rt.cpan.org

作为一个存储库，成千上万的作者通过完成自己的项目为CPAN做出了贡献，另外CPAN还有一个bug跟踪器。每个模块在Request Tracker (<https://rt.cpan.org>) 中都有自己的一个队列。这是报告模块问题的默认方式。

### 其他bug跟踪

一些模块作者更愿意使用其他跟踪方式而不是<https://rt.cpan.org>。可以查看模块文档，看看模块作者想要什么。模块作者有时会在模块文档中包含一些说明，不过有时则没有任何说明。由于安装时会留下README和META.yml等文件，在某个CPAN搜索网站查看这些文件可能会有帮助。

### perlbug

如果需要报告一个模块中由于perl本身带来的一个bug，可以使用perlbug工具。这个工具会收集你的平台和解释器的有关信息，使诊断bug的人能够得到他们需要的信息。这实际上只是一个界面，它将特殊格式的email消息发送到perlbug@perl.org，如果你愿意，也可以直接向这个地址发送邮件。这些报告会自动发送到Perl 5 Porters。有些模块有双重性，它们在标准库和CPAN上都有出现，所以要确定正确的位置可能稍有些难度。不过不要因此妨碍你报告问题。我们肯定能解决。

### rt.perl.org

perlbug将报告发送到位于<https://rt.perl.org>的请求跟踪器（Request Tracker），在这里你还可以了解现有的bug，包括标准库中模块的bug。如果在<https://rt.cpan.org>中找不到你想要的东西，还可以查看这个网站。

# 安装CPAN模块

人们在CPAN模块中主要使用两个构建系统。一个是基于常用的`make`工具来构建，另一个是利用纯Perl构建。

## 手动安装

人们通常不会手动安装CPAN模块，因为如果要手动安装，他们必须自己处理所有依赖关系，工作量很大。不过，手动安装也是可以的，了解如何手动安装会很有用。

查看模块时，你会找到一个`Makefile.PL`或`Build.PL`。可以采用同样的方式使用这两个文件，如表19-1所示。

表19-1：两个主要构建工具的构建命令

Makefile.PL	Build.PL
% perl Makefile.PL	% perl Build.PL
% make	% ./Build
% make test	% ./Build test
% make install	% ./Build install

基于默认值，这两个构建系统都会在你（或其他人）设置的库路径中安装模块，库路径是构建系统构建和安装`perl`二进制版本时建立的（你要用这个`perl`二进制版本运行构建文件）。可以在`perl -V`输出的末尾看到这些目录（库路径）。

你可能没有权限写这些目录，不过只要告诉构建文件把模块安装在哪里，就可以在你喜欢的任何目录中安装模块。构建文件会根据命令行选项或环境变量改变其行为。

```
% perl Makefile.PL INSTALL_BASE=/some/other/directory
```

```
% perl Build.PL --install_base /some/other/directory
```

如果已经在适当的环境变量中设置了必要的选项，就不用每次再指定选项。每个构建系统都有一个环境变量来保存默认的命令选项。在`/bin/sh`环境中可以如下所示：

```
% export PERL_MM_OPT='INSTALL_BASE=/some/other/directory'
% export PERL_MB_OPT='--install_base /some/other/directory'
```

使用采用`csh`语法的shell可以如下所示：

```
% setenv PERL_MM_OPT 'INSTALL_BASE=/some/other/directory'
% setenv PERL_MB_OPT '--install_base /some/other/directory'
```

不论使用哪一种方法告诉构建文件你希望将模块安装在哪里，它都会在你给定的路径的末尾附加`lib/perl5`<sup>注6</sup>。要记住这一点。

如果在另一个目录中安装模块，记得要告诉程序在哪里查找那些模块，可以使用`-I`开关：

```
% perl -I/some/other/directory/lib/perl5 program.pl
```

或者使用`PERL5LIB`环境变量：

```
% export PERL5LIB=/some/other/directory/lib/perl5
% perl program.pl
```

还可以在程序中使用`lib pragma`：

```
use lib qw(/some/other/directory/lib/perl5);
```

如果不记得这些路径，可以使用CPAN中的`local::lib`模块（这个模块不随标准Perl版本发布）。加载后，它会告诉你使用什么值。默认地，它会使用你的主目录下的子目录：

```
% perl -Mlocal::lib
export PERL_LOCAL_LIB_ROOT="/home/amelia";
export PERL_MB_OPT="--install_base /home/amelia/perl5";
export PERL_MM_OPT="INSTALL_BASE=/home/amelia/perl5";
export PERL5LIB="/home/amelia/perl5/lib/perl5/darwin-2level:/home/amelia/perl5/
lib/perl5";
export PATH="/Users/amelia/perl5/bin:$PATH";
```

或者你可以指定另一个目录：

```
% perl -Mlocal::lib=/some/other/directory
export PERL_LOCAL_LIB_ROOT="/some/other/directory";
export PERL_MB_OPT="--install_base /some/other/directory";
export PERL_MM_OPT="INSTALL_BASE=/some/other/directory";
export PERL5LIB="/some/other/directory/lib/perl5/darwin-2level:/some/other/
directory/lib/perl5";
export PATH="/some/other/directory/bin:$PATH";
```

你还必须自行建立这个环境，不过在程序中简单地使用`local::lib`就可以为你建立这个环境：

```
use local::lib;

use local::lib qw(/some/other/directory);
```

## CPAN客户程序

大多数人都用一个客户程序来安装模块<sup>注7</sup>。有3个流行的CPAN客户程序，分别针对有不同需求的不同用户而设计。你不必一直都使用同一个客户程序，这不是一个一生的选择。

---

注6： 这实际上是默认值。可以在编译`perl`时用`Configure`的`-Dinstallstyle`改变这个设置。

注7： 或者使用他们的操作系统提供的一个包管理器。



## cpan

*cpan*命令由标准库和CPAN.pm模块提供，它为安装模块提供了一种快捷的方式。只需要在命令行上指定你想要的模块：

```
% cpan IO::Interactive AnyEvent
```

要把模块安装在另一个不同的目录中，可以指定相应的配置。如果没有参数，*cpan*就会带你进入CPAN.pm shell：

```
% cpan
cpan> o conf makepl_arg INSTALL_BASE=/some/other/directory
cpan> o conf mbuild_arg "--install_base /some/other/directory"
cpan> o conf commit
```

你也可以启动CPAN.pm shell：

```
% perl -MCPAN -e shell
cpan> install POE
```

或者通过local::lib来使用：

```
% perl -MCPAN -Mlocal::lib -e shell
cpan> install Set::CrossProduct
```

## cpanp

Perl还提供了另一个CPAN界面，即CPANPLUS。这个项目希望从CPAN.pm的开发吸取教训，并重新开始：

```
% cpanp -i IO::Interactive AnyEvent
```

也可以启动CPANPLUS shell：

```
% perl -MCPANPLUS -e shell
CPAN Terminal> install POE
```

CPANPLUS使用了一个菜单驱动的配置系统，所以一旦进入这个shell，只要按其提示完成操作就可以。

## cpanminus

第3个流行的客户程序是*cpanminus*，或者就称为*cpanm*，如果你接受它的默认设置，可能就会喜欢这个客户程序。这是最小的一个客户程序，尽量满足大多数人的需要。它还默认地使用了local::lib。大多数人都很喜欢这个客户程序，它通常是一个不错的选择，除非你需要更复杂的功能。

由于`cpanm`力求易于使用，所以使用这个工具并不要求你安装其他模块<sup>注8</sup>。只需要下载这个工具，然后就可以开始使用了。`cpanm`文档展示了如何使用`curl`<sup>注9</sup>下载这个客户程序，然后将这个目录管道输出到`perl`，将它转换到`cpanm`。这是首选的方式，因为`cpanm`可以从你实际使用的`perl`二进制版本得到配置：

```
% curl -L http://cpanmin.us | perl - App::cpanminus
```

或者，也可以下载并保存为`cpanm`，然后运行。在Unix上，这（基本上）等同于保存结果并建立可执行文件，不过，在这里它使用`/usr/bin/env`来找出你的路径中的第一个`perl`：

```
% cd ~/bin
% curl -LO http://xrl.us/cpanm
% chmod +x cpanm
```

一旦有了`cpanm`，就可以让它安装模块了，如下所示：

```
% cpanm HTML::Barcode
```

## 创建CPAN模块

这里只是创建CPAN模块的一个简短介绍。如果详细介绍这个内容，需要整本书才能说清楚<sup>注10</sup>。《Intermediate Perl》（O'Reilly的Perl教程之一）更详细地介绍了这个内容。

## 开始创建模块

由于CPAN很早以前就已经开始运作，关于如何使用CPAN的最佳实践和标准约定已经形成，所以，对于一个好的模块需要有些什么，现在几乎所有人都已经达成了共识。如果使用某些工具来为你创建模块的骨架，你就不用一切都从头开始。

### h2xs

这个规范的模块创建工具实际上并不是一个模块创建工具。顾名思义，这个模块原本设计为要将C头文件转换为XS文件（XS是连接Perl和C的粘合语言）。这个工具取得了很大发展，甚至于如今大多数人使用它并不是为了得到它的主要特性（即转换文件），而是要用来创建模块：

---

注8：有些人认为标准库实际上就是一个启动工具包，使你能够运行`cpan`或`cpanp`。我们会看到它在将来的版本中如何演变，这也是那些无聊Perl会议永恒的话题之一。你可以在会上简单提一句，然后静静地坐在一旁看别人争论。

注9：`curl`是一个用来传输数据的命令行工具（<http://curl.haxx.se>）。

注10：确实有一本书专门介绍如何创建CPAN模块：Sam Tregar的《Writing Perl Modules for CPAN》，由Apress出版。

```
% h2xs -XAn Some::Module
Defaulting to backward compatibility with perl 5.14.2

Writing Some-Module/lib/Some/Module.pm
Writing Some-Module/Makefile.PL
Writing Some-Module/README
Writing Some-Module/t/Some-Module.t
Writing Some-Module/Changes
Writing Some-Module/MANIFEST
```

## Distribution::Cooker

`Distribution::Cooker`模块是最简单的模块创建工具，专门为要求不多的人们而设计<sup>注11</sup>。它会加工一个模板目录，这意味着你可以按你喜欢的任何方式设计你的模块，然后复制这个模块。一旦满足你的要求，不需要每次去修改其他工具的输出。实际上，要使用这个工具，最好的方法是用另一个工具启动，修改输出，直到你满意所得到的结果，然后设计一个相应的模板。

## Module::Starter

有些人不知道自己想要什么，对他们来说，`Module::Starter`是最好的选择。可以从`${HOME}/.module-starter/config`中的一个配置文件开始，这样你就无需输入太多内容：

```
author: Amelia Camel
email: amelia@example.com
builder: Module::Build
verbose: 1
```

然后运行`module-starter`，你会得到一个基本的模块结构：

```
% module-starter --module=Some::Module2
Created Some-Module
Created Some-Module/lib/Some
Created Some-Module/lib/Some/Module2.pm
Created Some-Module/t
Created Some-Module/t/pod-coverage.t
Created Some-Module/t/pod.t
Created Some-Module/t/manifest.t
Created Some-Module/t/boilerplate.t
Created Some-Module/t/00-load.t
Created Some-Module/ignore.txt
Created Some-Module/Build.PL
Created Some-Module/Changes
Created Some-Module/README
Created Some-Module/MANIFEST
Created starter directories and files
```

注意测试文件`Some-Module/t/boilerplate.t`。要在这里检查是否改变了某些默认设置，如模块的描述。

---

注11：这本书之所以会介绍`Distribution::Cooker`，只是因为它是本书作者之一编写的。



## Dist::Zilla

`Dist::Zilla`是一个复杂的工具，除了简单地创建初始模块外，它还会做很多其他工作。它会管理模块的整个生命周期，从概念到发布、测试、bug修正和重新发布的全过程。这有些太复杂了，我们没有时间详细解释，不过很多人非常喜欢这个工具。

## 测试模块

Perl的测试文化是它最吸引人的特性之一。我们已经介绍过CPAN Testers，他们会在各种平台上测试所有CPAN模块。作为一个模块作者，你要创建你自己的测试。我们打算告诉你有关的所有内容，因为这方面的内容在其他一些书中已经做了详尽的介绍，如《Intermediate Perl》和《Perl Testing: A Developer's Notebook》。

## 内部测试

如果你使用上述的两个标准模块构建工具，这意味着你已经有了一个测试工具。可以通过`test`运行测试目标：

```
% make test
```

```
% ./Build test
```

它们的工作是一样的：查找一个`test.pl`文件或一个`t/`目录。使用`test.pl`文件是老办法，这只是一个文件包含所有测试。使用`t/`子目录是一种更好的方法，因为它能包含多个测试文件，每个测试文件都以`.t`结尾，这个测试工具会运行所有子测试。

每个测试文件就是一个Perl程序，就像使用`Test::More`来完成工作。下面是一个示例测试文件，它会加载你的`Math::MySum`模块，并测试其`my_sum`方法：

```
use strict;
use warnings;
use Test::More;

BEGIN { use_ok( "Math::MySum" ) }
can_ok( "Math::MySum", "my_sum" );

my($i, $j) = (1, 3);
my $string = "Amelia";

is($i + $j, Math::MySum->my_sum( $i, $j ),
"Sum of $i and $j is 4");

like($string, qr/mel/, "String has mel in it");

done_testing;
```

这些程序会输出TAP（Test Anywhere Protocol），这是Larry发明的一个简单格式<sup>注12</sup>，并得到了其他人的扩展。这个程序的TAP输出如下：

```
ok 1 - use Math::MySum;
ok 2 - Math::MySum->can('my_sum')
ok 3 - Sum of 1 and 3 is 4
ok 4 - String has mel in it
1..4
```

还可以使用**blib**模块单个地运行测试，将构建库自动增加到@INC：

```
% perl -Mblib t/failingtest.t
```

另外可以使用**prove**工具：

```
% prove -vb t/failingtest.t
```

对于所有测试套件来说，难点在于要测试所有代码。由于你既是模块的作者，又是其测试的作者，就有可能有意无意地不覆盖那些困难的部分，从而能很容易地通过测试。为了检查这一点，**Devel::Cover**模块提供了一个**cover**程序，可以用这个程序来考察你的测试覆盖度：

```
% cover -test
```

**cover**命令为你运行测试套件，收集统计信息，并生成一个报表：

```
Reading database from ./cover_db
```

File	stmt	bran	cond	sub	time	total
blib/lib/Some/Module.pm	82.9	50.0	27.3	92.3	83.0	72.7

```
Writing HTML output to ./cover_db/coverage.html ...
done.
```

它会度量4种覆盖度：

**statement**

运行每一个语句。

**branch**

测试每一个分支，如有多个**elsif**块的**if**中，每个**elsif**都会计为一个单独的分支。

**condition**

如果有多个可能的条件，测试每一个条件组合。例如，有以下**if**条件：

---

注12：很多其他语言也包含了TAP。不要求TAP生产者与TAP消费者采用相同的语言。

```
if ($m && $n) { ... }
```

这个if有3个可测试的组合：\$m和\$n可能分别为true；\$m可能为false，在这种情况下\$n是什么都不重要；\$m可能为true，\$n可能为true或false。要分别测试这几种情况。

### *subroutine*

运行每一个子例程，这也是测试每一条语句的一部分。

## 外部测试

如果将你的模块上传到PAUSE，CPAN Testers会自动下载这个模块，进行测试，并向你发回结果。要在你没有的平台和Perl版本上完成测试，这样会很方便。你不用为此做所需的任何特殊准备。

不过，这只适用于公共模块。如果不打算把你的作品发布到CPAN，也可以做一些外部测试，为此可以建立你自己的CPAN Testers系统和一组测试机器。与常规CPAN Testers类似，你也使用同样的工具，不过要从你的私有来源获取模块。

还可以把Perl测试集成到很多连续集成测试框架中，如*smolder*（特别为Perl开发，但不限于此），Hudson、Jenkins或TeamCity。理解TAP的任何工具（现在有很多）都能分析你的测试输出。





# Perl的文化





不论你面对的是坐在键盘前敲命令的用户，还是通过网络发送信息的用户，都要对进入你的程序的数据倍加小心。程序另一端的用户可能会（恶意或无意地）向你发送一些有害无益的数据。Perl提供了一种特殊的安全检查机制，称为污染模式（taint mode），主要作用是隔离那些污染数据，以免用它们去做你不打算做的事情。例如，如果你误信了一个被污染的文件名，可能会把某个记录追加到你的口令文件，而你还以为只是把它追加到一个日志文件中。污染机制将在下一节“处理不安全的数据”中详细介绍。

在多任务环境中，一些“看不见的演员”在后台的动作可能会影响你的程序的安全性。如果你以为你对外部对象（尤其是文件）有绝对的所有权，好像你的程序是系统上的唯一进程一样，你就会遭遇一些错误，与直接处理数据或可疑来源代码带来的错误相比，这些错误更为微妙。Perl在这方面会有一些帮助，能检测出有些情况已经超出了你的控制范围，不过对于你能控制的情况，关键要了解哪些方法能防范那些看不见的捣乱者。后面的“处理计时问题”一节将讨论这些问题。

如果你从一个陌生人那里得到的数据是一段要执行的源代码，与得到数据相比，你要更加小心。Perl提供了一些检查，可以截获那些冒充为数据的隐蔽代码，以免你无意中执行这些代码。不过，如果确实想执行外部代码，可以使用Safe模块，它允许你隔离可疑的代码，在这个“隔离室”中代码无法做任何有害的事情，只能做一些有益的工作。这些内容将在本章后面“处理不安全的代码”一节讨论。

## 处理不安全的数据

即使用户可能比程序本身还不可靠，不过使用Perl可以很容易地保证程序安全。也就是说，有些程序需要向用户授予有限的权限，但不能放开其他权限。在Unix上，Setuid和

setgid程序就属于这一类，支持这种概念的其他操作系统上也有一些程序需要在多种特权模式下运行，那些程序也要求为用户授予权限。即使在不支持这种概念的操作系统上，这个原则也适用于网络服务器以及由这些服务器运行的所有程序（如CGI脚本、邮件列表处理器，以及/etc/inetd.conf中所列的守护程序）。所有这些程序都要求更高的安全性。

甚至从命令行运行的程序有时也很适合采用污染模式，特别是将以特权用户运行时。如果程序要处理不可信的数据，如从日志文件生成统计数据，或者使用LWP::\*或Net::\*来获取远程数据，运行这些程序时就应该显式地打开污染模式；不谨慎的程序往往会有沦为“特洛伊木马”的风险。既然这种冒险不会让程序感到丝毫兴奋，那就没有理由不小心一点。

Unix命令行shell实际上只是调用其他程序的框架，与shell相比，利用Perl编写安全的程序会很容易，因为它很直接，而且是自包含的。大多数shell编程语言的前提都是对单行脚本完成多次神秘的替换，与之不同，Perl使用了一种更传统的计算机制，隐藏的小“机关”更少。另外，由于Perl语言提供了更多的内置功能，它很少依赖于外部（可能不可信的）程序来实现其目的。

在Unix中（Perl的发源地），要想破坏系统的安全性，首选的方法是引诱一个特权程序做它原本不该做的事情。要避免这种攻击，Perl开发了一种独特的方法来应对这种危险环境。只要检测到运行程序的真实用户（或组）ID与有效用户（或组）ID不同，Perl就会自动启用污染模式<sup>注1</sup>。尽管包含Perl脚本的文件没有设置setuid或setgid位，这个脚本仍可能以污染模式执行。如果你的脚本由另一个程序调用，而运行那个程序的真实ID与有效ID不同，就会发生这种情况。如果一个Perl程序没有设计为在污染模式下操作，那么一旦发现违反污染策略，就会提前终止。这样是有道理的，因为以前就是采用这种伎俩对shell脚本动手脚来破坏系统的安全性。Perl可没有那么容易上当。

还可以用-T命令行开关显式地启用污染模式。对于守护程序、服务器以及代表别人运行的任何程序（如CGI脚本），都应当这么做。如果程序要远程运行，或者会由网络上某个人匿名运行，执行这些程序的环境最为危险。不要害怕偶尔说“不！”与通常的看法不同，你可以非常小心谨慎，别人不会因此认为你过于刻板。

在更为关心安全的网站上，所有CGI脚本都要在-T标志下运行，这不只是一个好主意，更应算是一条法令。我们并不是说在污染模式中运行就足以保证脚本的安全。这还不够，有关的方方面面需要整本书来介绍。不过，如果未在污染模式下运行你的CGI脚本，这意味着你已经放弃了Perl为你提供的最强保护。

---

注1： Unix权限中的setuid位是04000，setgid位是02000，可以设置其中任意一位（或者二者都设置），为程序用户授予程序所有者的一些权限（它们统称为set-id程序）。其他操作系统可以采用其他方式为程序授予特殊权限，不过原则是一样的。



在污染模式中，Perl采取了一些特殊的防范措施来避免一些明显和微妙的陷阱，这称为污染检查（taint checks）。其中一些检查相当简单，如验证以确保没有设置危险的环境变量，以及路径中的某些目录不允许别人写，细心的程序员总会使用类似这样的检查。不过，另外一些检查由语言本身支持，正是基于这些检查，我们可以建立比相应C程序更为安全的特权Perl程序，或者与使用不提供污染检查的其他任何语言编写的脚本相比，Perl CGI脚本会更为安全（就我们所知，除了Perl之外，所有其他语言都没有提供污染检查）。

道理很简单：不能使用程序之外的数据来影响程序之外的东西，至少不能无意中影响。来自程序之外的所有数据都标为被污染的数据，包括所有命令行参数、环境变量和文件输入。污染数据不能在调用子shell的任何操作中直接或间接使用，也不能在修改文件、目录或进程的操作中使用。如果某个表达式之前引用了一个被污染的值，这个表达式中设置的任何变量本身也会被污染（即使从逻辑上讲被污染的值不可能影响这个变量）。不过，使用一个被污染的变量来选择一个未被污染的值，这不会污染结果。例如，下面的\$value不会被污染：

```
my $value = $tainted ? 'Amelia' : 'Camelia'; # $value不被污染
```

甚至以下情况中也不会污染：

```
my $value = do {  
    if( $tainted ) { 'Amelia' }  
    else           { 'Camelia' }  
};
```

由于污染性质只与各个标量关联，可能数组或散列中的某些单个值被污染，而另外一些未被污染（不过，只可能污染散列中的值，而不会污染键。有关内容稍后介绍）。

下面的代码展示了如果按顺序执行所有这些语句污染是如何发生的。标为“不安全”的语句会触发一个异常，而标为“OK”的语句不会触发异常。

```
$arg = shift(@ARGV);          # $arg现在被污染（由于@ARGV）  
$hid = "$arg, 'bar'";         # $hid也被污染（由于$arg）  
$line = <>;                   # 被污染（读自外部文件）  
$path = $ENV{PATH};           # 被污染（由于%ENV），不过见下面的代码  
$mine = "abc";                # 未被污染  
  
system "echo $mine";           # 不安全，直到设置PATH  
system "echo $arg";            # 不安全，用被污染的$arg使用sh  
system "echo", $arg;           # 一旦设置PATH则OK（未使用sh）  
system "echo $hid";            # 两方面都不安全：污染，PATH  
  
$oldpath = $ENV{PATH};         # $oldpath被污染（由于%ENV）  
$ENV{PATH} = "/bin:/usr/bin"; # （OK，可以执行其他程序）  
$newpath = $ENV{PATH};        # $newpath未被污染  
  
delete @ENV{qw{IFS  
                CDPATH
```



```

ENV
BASH_ENV}};    # 使%ENV更安全

system "echo $mine";    # 正确，一旦重新设置路径则安全
system "echo $hid";    # 不安全（由于被污染的$hid）

open(00F, "< $arg");    # 正确（不检查只读open）
open(00F, "> $arg");    # 不安全（试图写被污染的arg）

open(00F, "echo $arg|")    # 不安全，由于被污染的$arg，不过
    || die "can't pipe from echo: $!";

open(00F, "-|")    # 认为OK：见下面的代码
    || exec "echo", $arg    # 可以用exec执行列表
    || die "can't exec echo: $!";

open(00F, "-|", "echo", $arg)    # 与前面相同，也认为OK
    || die "can't pipe from echo: $!";

$shout = `echo $arg`;    # 不安全（由于被污染的$arg）
$shout = `echo abc`;    # $shout被污染（由于反引号）
$shout2 = `echo $shout`;    # 不安全（由于被污染的$shout）

unlink $mine, $arg;    # 不安全（由于被污染的$arg）
umask $arg;    # 不安全（由于被污染的$arg）

exec "echo $arg";    # 不安全（由于被污染的$arg传入shell）
exec "echo", $arg;    # 认为OK（不过见下面的代码）
exec "sh", "-c", $arg;    # 认为OK，不过实际上不完全如此

```

如果试图做一些不安全的事情，会得到一个异常（除非捕获该异常，否则它会成为一个致命错误），如“Insecure dependency”（不安全的依赖关系）或“Insecure \$ENV{PATH}”（不安全的\$ENV{PATH}）。参见本章后面“清理环境”一节。

如果向system、exec或管道式open传入一个LIST，将不会检查参数是否被污染，因为对于一个参数LIST，Perl并不需要调用可能有危险的shell来运行命令。采用这种LIST形式，仍然很容易地可以写一个不安全的system、exec或管道式open，如上面最后一个例子所示。这些形式不需要检查，因为Perl认为你使用这些形式时知道自己在做什么。

不过，有时你不知道传递了多少参数。如果为这些函数提供一个数组<sup>注2</sup>，其中只包含一个元素，这就好像你传入了一个字符串，所以可能会使用shell。解决方法是在间接对象槽中传入一个显式的路径：

```

system @args;    # 除非@args == 1，否则不会调用shell
system { $args[0] } @args;    # 即使列表只有一个参数，也会绕过shell

```

## 检测和清洗污染数据

要测试一个标量变量是否包含污染数据，可以使用以下is\_tainted函数。它利用了这样一

注2： 或者生成一个列表的函数。

个事实：如果试图编译污染数据，`eval STRING`会产生一个异常。尽管要编译的表达式中使用的`$nada`变量总为空，但这并不重要；如果`$arg`被污染，它也会被污染。外层`eval BLOCK`不完成任何编译。它的作用只是捕获对内层`eval`提供污染数据时产生的异常。如果`eval`产生一个异常，那么`$@`变量肯定非空，否则不为空，所以我们可以测试其长度是否为0，并返回这个结果：

```
sub is_tainted {
    my $arg = shift;
    my $nada = substr($arg, 0, 0);    # 长度为0
    local $@;                        # 保留调用者的版本
    eval { eval "# $nada" };
    return length($@) != 0;
}
```

Perl提供的`Scalar::Util`模块已经为你完成了这个工作，它提供了一个`tainted`函数：

```
use Scalar::Util qw(tainted);
print "Tainted!" if tainted( $ARGV[0] );
```

`Taint::Util` CPAN模块则更进一步。它有一个可以做同样事情的`tainted`函数，另外还有一个`taint`函数，可以污染任何数据：

```
use Taint::Util qw(tainted taint);

my $scalar = 'This is untainted';    # 未被污染
taint( $scalar );                    # 现在被污染
```

想用污染数据进行测试时，这对于编写测试脚本会很方便：

```
use Test::More;
use Taint::Util qw(tainted taint);

my $tainted = 'This is untainted';    # 未被污染
taint( $tainted );                    # 现在被污染

ok( tainted( $tainted ), 'Data are tainted' );
is( refuse_to_work( $tainted ), undef, 'Returns undef with tainted data' );

done_testing();
```

不过有关污染与否的测试就到此为止。通常你很清楚哪些变量包含污染数据，你只是想要清除这些数据的污染性质。要想绕过污染机制，正式的方法只有一个：引用之前一个正则表达式匹配返回的子匹配<sup>注3</sup>。如果写了一个包含捕获小括号的模式，可以通过匹配变量（如`$1`、`$2`和`$+`）访问捕获的子串，或者在列表上下文中计算模式来访问。不论采用哪一

---

注3： 还有一个不太正式的方法，可以存储被污染的字符串作为一个散列的键，并获取这个键。由于键实际上并不完全是SV（内部标量值），它们不会携带污染性质。这种行为将来有可能会改变，所以不要依赖这种做法。处理键时要当心，以免无意地清洗了数据并对它们做了不安全的处理。

种方法，前提是你写这个模式清除危险的东西时，要很清楚自己在做什么。所以你要真正的思考，绝不要盲目地清洗，否则你要应对整个污染机制。

最好验证变量只包含“好”字符，而不是检查它是否包含“坏”字符。这是因为，很容易漏掉你没有想到的坏字符。例如，下面给出一个测试，想要确保\$string只包含“单词”字符（字母、数字和下划线）、连字号、@符号和点号：

```
if ($string =~ /^([-@\w.]+)$/) {  
    $string = $1;          # $string 现在未被污染  
}  
else {  
    die "Bad data in $string";    # 记录在某个地方  
}
```

这样得到的\$string可以安全地用在外部命令中，因为/\w+/正常情况下不匹配shell元字符，也不匹配对shell有任何特殊含义的其他字符<sup>注4</sup>。如果我们使用/(.+)/s，可能就不安全了，因为这个模式允许任意字符通过。不过Perl不会进行检查。完成清洗时，一定要对模式特别当心。使用正则表达式清洗数据是Perl唯一允许的清洗脏数据的内部机制。有时这个方法是完全错误的。如果由于运行set-id而不是因为有意打开-T而进入污染模式，可以派生一个权限更低的子进程来降低风险，参见本章后面“清理环境”一节。

use re 'taint' pragma会对当前词法作用域结束之前的所有模式匹配禁用隐式清洗。如果你只想从一些可能被污染的数据抽取一些子串，可以使用这个pragma，不过，由于你没有仔细考虑安全性，最好保留这些子串的污染性质，以免以后发生不幸的事件。

假设你要匹配以下模式，这里\$fullpath已被污染：

```
($dir, $file) = $fullpath =~ m!(.*/)(.*)!s;
```

默认地，\$dir和\$file现在未被污染。不过你可能不想这么“武断”，因为你从来没有考虑过安全问题。例如，如果\$file包含字符串“; rm -rf \* ;”，你可能不会接受，这里只是举一个比较极端的例子。如果\$fullpath被污染，下面的代码会保留两个结果变量也是污染的：

```
{  
    use re "taint";  
    ($dir, $file) = $fullpath =~ m!(.*/)(.*)!s;  
}
```

一个好的策略是，默认地让整个源文件中的子匹配都被污染，只是在嵌套范围中根据需要选择地允许清洗：

---

注4：除非你使用一个有意破坏的本地化环境。Perl假设你的系统的本地化环境定义可能被破坏。

因此，在locale pragma下运行时，包含有符号字符类的模式（如\w或[:alpha:]）会生成污染的结果。



```

use re "taint";
# 文件的其余部分中保持$1等是污染的
{
    no re "taint";
    # 这个块现在清洗re匹配
    if ($num =~ /\^(\\d+)$/) {
        $num = $1;
    }
}

```

来自文件句柄或目录句柄的输入会自动被污染，除非它来自特殊文件句柄DATA。如果你愿意，也可以通过IO::Handle模块的untaint函数标志其他句柄为可信源：

```

use IO::Handle;

IO::Handle::untaint(*SOME_FH);      # 采用过程方式，
SOME_FH->untaint();                 # 或者采用面向对象方式

```

关闭整个文件句柄的清洗会有很大风险。你真的知道它是安全的吗？如果你要这么做，起码要验证除了文件的所有者以外其他人都不能写这个文件<sup>注5</sup>。如果你使用的是一个Unix文件系统（而且谨慎地限制只有超级用户才能执行chown(2)），可以使用下面的代码：

```

use File::stat;
use Symbol "qualify_to_ref";
sub handle_looks_safe(*) {

    my $fh = qualify_to_ref(shift, caller);
    my $info = stat($fh);
    return unless $info;

    # 所有者既不是超级用户也不是"我",
    # 真实uid在$<变量中
    if ($info->uid != 0 && $info->uid != $<) {
        return 0;
    }

    # 检查组或其他人是否可以写文件。
    # 另外使用066检测可读性
    if ($info->mode & 022) {
        return 0;
    }

    return 1;
}

use IO::Handle;
SOME_FH->untaint() if handle_looks_safe(*SOME_FH);

```

要对文件句柄而不是文件名调用stat，以避免危险的竞态条件。参见本章后面“处理竞态条件”一节。

---

注5： 尽管也可以清洗目录句柄，但这个函数只作用于文件句柄。这是因为，给定一个目录句柄，没有一种可移植的方式能够将它的文件描述符抽取到stat。

需要说明，这个例程只是一个很好的开始。更“执着”的版本还会检查所有父目录，尽管不能对目录句柄使用stat。不过，如果某个父目录是公共可写的，不论是否有竞态条件，你都应该知道会有麻烦了。

对于哪些操作有危险，Perl有自己的想法，不过另外还有一些不关心是否使用污染值的操作，使用这些操作时还是有可能遇到麻烦。只注意输入还不够。Perl的输出函数不会测试参数是否被污染，不过有一些环境中，参数是否被污染是有区别的。如果不仔细考虑输出，最后输出的字符串对于那些处理输出的人来说可能含义完全改变。如果在一个终端上运行，特殊的转义和控制码可能会导致浏览器的终端表现异样。如果在一个web环境中，只是轻率地返回交给你的所有数据，可能会在不知情的情况下生成HTML标记，这可能会彻底改变页面的外观。更糟糕的是，有些标记甚至还会在浏览器上执行代码。

设想一个常见的情况，假设有一个留言簿，访问者可以输入消息，其他人调用时会显示这些消息。一个恶意访问者可能会输入难看的HTML标记，甚至加入<SCRIPT>...</SCRIPT>序列（这会在后续访问者的浏览器中执行代码（如JavaScript））。

检查污染数据（访问你自己的系统资源）时，要很小心地检查其中只包含“好”字符，同样的，在web环境中显示用户提供的数据时也要小心。例如，要去除数据中不好的字符（即“好”字符列表中未指定的任何字符），可以使用如下代码：

```
$new_guestbook_entry =~ tr[_a-zA-Z0-9 ,./!()?@+*-][ ]dc;
```

当然你不会用它来清理一个文件名，因为你可能不希望文件名中包含空格或斜线（只针对初学者）。不过，这对于保护留言簿不会偷偷潜入HTML标记和实体已经足够了。每个数据清洗情况都稍有些区别，所以一定要花些时间来确定哪些是允许的，哪些不允许。污染机制的目的是捕获愚蠢的错误，而不是完全不加思考。

## 清理环境

从你的Perl脚本执行另一个程序时，不论如何执行，Perl都会进行检查以确保你的PATH环境变量是安全的。由于它来自于你的环境，所以PATH一开始就是污染的。所以，如果你想运行另一个程序，Perl会生成一个“Insecure \$ENV{PATH}”异常。将PATH设置为一个已知未污染的值时，Perl会确保该路径中的每个目录除了该目录的所有者和组以外其他任何人都不可写。否则，它会产生一个“Insecure directory”异常。

你可能会奇怪地发现，即使你指定了想要执行的命令的完全路径名，Perl也会关心PATH。对于一个绝对文件名，不会使用PATH来查找要运行的可执行文件，这一点不假。不过，没有理由相信你运行的那个程序不会突然“转身”去执行另外一个程序，然后由于不安全的PATH而遇到麻烦。所以在你调用任何程序之前，不论以何种方式调用，Perl都会强制你设置一个安全的PATH。



可能带来麻烦的环境变量并不只有PATH。由于一些shell使用变量IFS、CDPATH、ENV和BASH\_ENV，Perl在运行另一个命令之前会确保这些变量都为空或者都未被污染。可以将这些变量设置为某个已知安全的值，或者将它们从环境中全部删除：

```
delete @ENV{qw(IFS CDPATH ENV BASH_ENV)}; # 使用%ENV更安全
```

正常环境中的一些便利特性在危险环境中可能会变成安全隐患。即使你记得不允许文件名中包含换行符，但重要的是要知道open不只访问命名文件。如果对文件名参数增加适当的装饰，单参数或两参数的open调用还可以通过多种手段运行任意的外部命令，如利用管道、派生当前进程的额外副本、复制文件描述符，以及将特殊文件名“-”解释为标准输入或输出的一个别名。它还可以忽略前导或末尾空白符，这样可以伪装这些有问题的参数，使它们躲过你的检查模式。尽管Perl的污染检查能捕获用于管道式open（除非你使用了一个单独的参数表）以及非只读文件open的污染参数，不过它生成的异常很可能会让你的程序表现失常。

如果你想使用任何外来数据作为要打开的文件名的一部分，至少要包含一个用空格分隔的显式模式。不过，可能最安全的做法是使用底层sysopen或三参数形式的open：

```
# 魔法open，可以是任何形式
open(FH, $file) || die "can't magic open $file: $!";

# 确保是只读文件open而不是一个管道
# 或fork，不过仍允许文件描述符和"-",
# 并忽略名字两端的空白符
open(FH, "< $file") || die "can't open $file: $!";

# WYSIWYG open: 禁用所有便利特性
open(FH, "<", $file) || die "can't open $file: $!";

# 与WYSIWYG 3参数版本性质相同
require Fcntl;
sysopen(FH, $file, O_RDONLY) || die "can't sysopen $file: $!";
```

这些步骤还不够好，Perl不会阻止你打开污染的文件名来读取数据，所以要仔细考虑显示给人们的信息。如果程序打开由用户提供一个任意文件名来读取数据，然后隐藏这个文件的内容，这仍然存在安全问题。如果这是一封私人信件呢？如果它是系统的口令文件呢？如果是工资信息或者你的股票投资信息呢？

对于一个可能有敌意的用户<sup>注6</sup>提供的文件名，在打开之前一定先要仔细检查。例如，可能要验证路径中没有隐藏的目录部分。类似“../..../..../..../etc/passwd”的名字就是这一类臭名昭著的例子。可以通过确保路径名中没有斜线（假设这是你的系统的目录分隔符）来保护自己。另一个常见的手段是在文件名中加入换行符或分号，一些不太好的命令行解释

---

注6： 在网络上，你唯一能相信没有敌意的人正是那些积极表现出敌意的人。



器可能会解释这些符号并上当受骗，在文件名中间启动一个新命令。正是因为这个原因，污染模式不允许运行未经检查的外部命令。

## 降权限访问命令和文件

下面的讨论只适用于类Unix系统的一些安全工具。其他系统的用户完全可以跳过这一节。

如果运行set-id，做那些危险的操作时要尽量以用户权限而不是程序权限来运行。也就是说，只要调用open、sysopen、system、反引号和任何其他文件或进程操作，都要保护自己，把有效UID或GID重新设置为真实UID或GID。在Perl中，对于setuid脚本可以写为\$> = \$<（或者如果你声明了use English，可以写为\$EUID = \$UID），对于setgid脚本，可以写为\$) = \$( (\$EGID = \$GID)。如果这两个ID都已设置，那么它们都需要重置。不过，有时这样并不可行，因为程序后面可能还需要那些提升的权限。

对于这些情况，Perl提供了一个很安全的方法，允许从一个set-id程序安全地打开文件或管道。首先，使用特殊的open语法派生一个子进程，用一个管道连接父进程和子进程。在子进程中，将用户ID和组ID重置为原来的值或已知安全的值。还可以修改子进程自己的任何属性，而不影响父进程，这允许你改变工作目录、设置文件创建掩码，或者调整环境变量。子进程不再在额外的权限下执行，它最后会调用open，并把它代表普通用户所能访问的所有数据传递给强大的父进程。

尽管向system和exec提供多个参数时不会使用shell，不过反引号操作符不支持这种调用约定。使用派生子进程技术，我们可以很容易地模拟反引号而不用担心shell转义，而且权限会降低（相应地也更安全）：

```
use English; # 使用$UID等
die "Can't fork open: $!" unless defined($pid = open(FROMKID, "-|"));
if ($pid) { # 父进程
    while (<FROMKID>) {
        # 做些处理
    }
    close FROMKID;
}
else {
    $EUID = $UID; # setuid(getuid())
    $EGID = $GID; # setgid(getgid()), 以及getgroups(2)上的initgroups(2)
    chdir("/") || die "can't chdir to /: $!";
    umask(077);
    $ENV{PATH} = "/bin:/usr/bin";
    exec "myprog", "arg1", "arg2";
    die "can't exec myprog: $!";
}
```

这是目前为止从set-id脚本调用其他程序的最佳方法。首先确保绝对不使用shell执行任何代

码，另外使用exec执行程序之前要降低权限（不过，由于列表形式的system、exec和管道式open都特别免除对其参数进行污染检查，所以还要当心传入的参数）。

如果不需要降低权限，只希望实现反引号或管道式open，而不想冒shell截获参数的风险，可以使用以下代码：

```
open(FROMKID, "-|") || exec("myprog", "arg1", "arg2")
|| die "can't run myprog: $!";
```

然后在父进程中读FROMKID。在Perl的v5.6.1版本中，可以写为：

```
open(FROMKID, "-|", "myprog", "arg1", "arg2");
```

除了从set-id程序运行命令，这种派生子进程的技术在其他方面也很有用。还可以利用这种技术以运行程序的用户ID打开文件。假设你有一个setuid程序，需要打开一个文件完成写操作。你不想在额外的权限下运行open，不过也不能永久地降低权限。所以可以安排一个派生的子进程副本，降低它的权限来为你完成open操作。等你想要写这个文件时，可以写至子进程，它再为你写文件。

```
use English;

defined ($pid = open(SAFE_WRITER, "|-"))
|| die "Can't fork: $!";

if ($pid) {
    # 你是父进程，将数据写至SAFE_WRITER子进程
    print SAFE_WRITER "@output_data\n";
    close SAFE_WRITER
        || die $! ? "Syserr closing SAFE_WRITER writer: $!"
        : "Wait status $? from SAFE_WRITER writer";
}
else {
    # 你是子进程，所以取消额外的权限
    ($EUID, $EGID) = ($UID, $GID);

    # 在原用户的权限下打开文件
    open(FH, "> /some/file/path")
        || die "can't open /some/file/path for writing: $!";

    # 从父进程（现在是stdin）复制到文件
    while (<STDIN>) {
        print FH $_;
    }
    close(FH) || die "close failed: $!";
    exit;      # 不要忘记撤销SAFE_WRITER
}
```

无法打开文件时，子进程会打印一个错误消息并退出。父进程写至现在已经“不存在”的子进程的文件句柄时，它会触发一个中断管道信号（SIGPIPE），除非捕获或忽略这个信号，否则这是致命的。参见第15章“信号”一节。

## 应对污染检查

污染模式作为一个开发工具，可以帮助你找到哪里需要净化数据。它并不能保证你的程序完全没有问题，仍然会有一些不好的事情发生。实际上对付它并不难。

`-T`命令行开关强制要求污染检查，可以把它放在你的shebang行上：

```
#!/usr/bin/perl -T

system 'echo', $ARGV[0];
```

如果从命令行用`perl`运行，而且没有`-T`，污染检查就会失败：

```
% perl echo.pl
"-T" is on the #! line, it must also be used on the command line
```

高水平的用户可能会打开污染模式，不过把平常致命的消息转换为警告。`-t`开关会打开污染模式，不过只警告冲突问题。`system`仍接受污染数据：

```
% perl -t echo.pl Amelia
Insecure $ENV{PATH} while running with -t switch
Insecure dependency in system while running with -t switch
Insecure $ENV{PATH} while running with -t switch
Amelia
```

作为`setuid`运行时（会自动打开污染模式），可以类似地用`-u`应对污染检查：

```
% perl -t echo.pl Amelia
Insecure $ENV{PATH} while running with -u switch
Insecure dependency in system while running with -u switch
Insecure $ENV{PATH} while running with -u switch
Amelia
```

类似地，`-U`开关允许`perl`运行“不安全”的操作，不过还需要指定`-T`：

```
% perl -TU echo.pl Amelia
Amelia
```

如果还希望得到警告，可以使用`-w`：

```
% perl -TU -w echo.pl Amelia
I Insecure $ENV{PATH} while running with -w switch
Insecure dependency in system while running with -w switch
Insecure $ENV{PATH} while running with -w switch
Amelia
```

程序员可以通过适当地不净化数据来应对污染模式，我们已经展示了一些方法。例如，如果有一个匹配所有字符的快捷方式：

```
my $untainted = $tainted =~ m/(.*)/;
```

代码审查时可能已经识别出它被污染，所以聪明人可能会通过一个散列传递数据。由于污



染只应用于标量变量，所以散列键不会被污染。使用标量变量作为散列键可以消除它的所有魔法（magic）：

```
my (untainted) = keys %{ { $untainted => 1 } };
```

需要考虑的方面还有很多。要了解更多技巧，参见《Mastering Perl》的“安全编程技术”一章。

## 处理计时问题

有时你的程序的行为会对超出你控制之外的外部事件的时间极其敏感。其他程序（特别是有敌意的程序）与你的程序竞争相同的资源（如文件或设备）时，往往就存在这个问题。在一个多任务环境中，无法预测等待运行的进程会按什么顺序得到授权来访问处理器。对于所有符合条件的进程，它们的指令流是交错的，所以第一个进程得到一些CPU时间，然后是另一个进程，如此继续。该轮到谁来运行以及允许它运行多长时间，这看起来都是随机的。如果只有一个程序，这不成问题。不过，如果有多个程序分享共同的资源，这就有问题了。

线程程序员对这些问题尤其敏感。他们很快就知道不能写这样的代码：

```
$var++ if $var == 0;
```

而应该写为：

```
{
    lock($var);
    $var++ if $var == 0;
}
```

如果有多个执行线程试图同时运行这个代码，前一种方法会生成不可预测的结果。如果把文件看作是共享对象，认为进程是竞争访问这些共享对象的线程，可以看到会出现同样的问题。毕竟，进程实际上就是一个有某种目的的线程。反之亦然。

计时的不可预测性不仅影响特权情况，也会影响非特权情况。我们首先介绍如何处理原先Unix内核中长期存在的bug，这些bug会影响所有set-id程序。然后再讨论一般的竞态条件，说明它们如何变成安全漏洞，并介绍可以采取哪些步骤避免落入这些漏洞。

## Unix内核安全bug

对于像shell这样灵活而神秘的解释器，为它们赋予特殊的权限会带来一些明显的问题，除此之外，老版本的Unix还存在一个内核bug，会使set-id脚本在进入解释器之前就已经不安全了。这个问题并不在于脚本本身，而是由于内核发现set-id可执行脚本时出现的一个竞态条件（内核不能识别#!的机器不存在这个bug）。内核打开这样一个文件来查看要运行

哪个解释器时，在这个解释器（现在是set-id）启动和重新打开文件之前会有一个延迟。这个延迟就为有恶意的人提供了一个机会，他们可以利用这个机会修改文件，特别是当你的系统支持符号链接时。

幸运的是，有时这个内核“特性”会被禁用。不过，有好几种禁用这个特性的方法。系统可以通过设置set-id位宣布脚本不合法，但这样帮助不大。或者，系统可以忽略脚本的set-id位。对于后一种方法，Perl注意到Perl脚本上的set-id位时，可以模拟setuid和setgid机制。这是通过一个特殊的可执行文件（名为suidperl）实现的，如果需要<sup>注7</sup>，会为你自动调用这个文件。不过，如果没有禁用内核set-id脚本特性，Perl会“大声抱怨”你的setuid脚本不安全。你要么需要禁用内核set-id脚本“特性”，要么需要在脚本外加一个C包装器。C包装器就是一个已编译的程序，除了调用你的Perl程序外，它什么也不做。已编译程序不会像set-id脚本那样受到内核bug的影响。

下面是一个用C编写的简单包装器：

```
#define REAL_FILE "/path/to/script"
main(ac, av)
    char **av;
{
    execv(REAL_FILE, av);
}
```

将这个包装器编译为一个可执行映像，然后构建这个可执行映像，而不是构建你的set-id脚本。一定要用一个绝对文件名，因为C还没有聪明到会对你的PATH完成污染检查。

另一个可能的方法是使用试验性的C代码生成器作为Perl编译器。脚本的已编译映像不会有竞态条件（参见第16章）。

近几年，开发商终于开始提供没有set-id bug的系统。在这些系统上，内核为解释器提供set-id脚本的名字时，不再使用容易被篡改的文件名，而是会传递一个表示文件描述符的特殊文件，如/dev/fd/3。这个特殊文件已经从脚本打开，所以不会存在恶意脚本可能利用的竞态条件<sup>注8</sup>。大多数现代版本的Unix都使用这个方法避免两次打开相同文件名所带来的竞态条件。

## 处理竞态条件

现在我们来讨论竞态条件（race conditions）。这到底是什么？竞态条件在安全性讨论中经

---

注7：不仅是需要，而且还要允许，如果Perl检测到脚本所在的文件系统安装（mount）时有nosuid选项，那么这个选项仍有效。不能以这种方式使用Perl偷偷绕过系统管理员的安全策略。

注8：在这些系统上，Perl要带-DSETUID\_SCRIPTS\_ARE\_SECURE\_NOW编译。构建perl的Configure程序会尝试自己来确定这个选项，所以你不用显式地指定。



常出现（当然它们在不安全的程序中出现更为频繁，真是很不幸）。这是因为竞态条件是滋生编程错误的肥沃土壤，而且这些错误经常会变成安全漏洞（这是一种委婉的说法，实际上是指摧毁安全性）。如果多个相互关联的事件的结果依赖于这些事件的顺序，但是由于不确定的计时问题，这个顺序无法保证，这种情况下就会出现竞态条件。每个事件都想竞争成为第一个完成的事件，系统的最终状态完全不可预测。

假设有一个进程在覆盖一个已有的文件，而另一个进程在读这个文件。你无法预出你读到的是老数据、新数据还是二者的混合产物。你甚至无法知道是否已经读到所有的数据。读进程可能在竞争中胜出，读取到文件末尾然后退出。与此同时，写进程在读进程达到文件末尾之后继续工作，文件将从读进程停止读的位置继续扩展，而读进程对此一无所知。

解决方法很简单：只需要让读写进程双方使用`flock`锁定文件。读进程通常请求一个共享锁，写进程一般请求排他锁。只要双方都请求并遵守这些建议锁，读写就不会交叉，也不会出现数据不完整的情况。参见第15章“文件锁定”一节。

每次对文件名的操作会控制该文件的后续操作时，你就会遭遇一种不太明显的竞态条件。如果应用于文件名而不是文件句柄，文件测试操作符将成为径直通向竞态条件的小径。考虑以下代码：

```
if (-e $file) {
    open(FH, "<", $file)
    || die "can't open $file for reading: $!";
}
else {
    open(FH, ">", $file)
    || die "can't open $file for writing: $!";
}
```

这段代码看起来很直接，不过它也面临竞争。不能保证`-e`测试返回的答案直到调用某一个`open`时仍有效。在`if`块中，在文件打开之前，可能已经有另一个进程删除了这个文件，这样一来，你就找不到你以为在那里的那个文件。在`else`块中，轮到第二个`open`创建文件之前，可能已经有一个进程创建了这个文件，所以你以为没有这个文件，但实际上这个文件已经存在了。简单的`open`函数会创建新文件但会覆盖原来已有的文件。你可能会认为你就是想要覆盖现有的文件，不过想想看，现有的文件可能是一个新创建的别名或者符号链接，指向系统上另外某个位置上的一个文件，而你并不想覆盖那个文件。也许你会说你 知道任何特定时刻一个文件名表示什么，不过只要有其他进程能够访问这个文件所在的目录，而且那些进程也在同一个系统上运行，你就不能完全肯定。

要修正这个覆盖问题，需要使用`sysopen`，它对创建一个新文件还是覆盖一个现有文件分别提供了单独的控制。我们放弃了那个`-e`文件存在性测试，因为它在这里没有任何用处，只会让我们更有可能遭遇竞态条件。

```
use Fcntl qw/O_WRONLY O_CREAT O_EXCL/;
```



```
open(FH, "<", $file)
|| sysopen(FH, $file, O_WRONLY | O_CREAT | O_EXCL)
|| die "can't create new file $file: $!";
```

在`open`失败后，而且在`sysopen`试图打开一个新文件来完成写操作之前，在此期间即使文件以某种方式出现，也不会有问题，因为基于所提供的标志，`sysopen`会拒绝打开一个已经存在的文件。

如果有人想哄骗你的程序做些不合适的操作，只需在你没有想到的时候让文件出现和消失，他就能抓住机会。要想减少被骗的风险，一种办法是承诺不要对一个文件名操作多次。一旦文件打开，就忘记文件名（除非可能还要用于错误消息），只处理表示这个文件的句柄。这样会安全得多，因为，尽管可能有人用你的文件名耍花招，但他不能对你的文件句柄动手脚（或者，如果他确实能，那也是因为你允许他这么做，参见第15章的“传递文件句柄”一节）。

在本章前面，我们介绍过一个`handle_looks_safe`函数，它会对文件句柄（而不是文件名）调用Perl的`stat`函数，来检查其所有者和权限。使用文件句柄对于正确操作至关重要，如果使用文件名，就不能保证我们检查的那个文件正是我们刚打开（或将要打开）的那个文件。一些可恶的家伙可能已经删除了我们的文件，并在`stat`和`open`之间迅速换成一个有恶意企图的文件。`stat`和`open`哪一个在前并不重要，这两个调用之间总有机会动手脚。你可能认为这个风险很小，因为时间窗口非常短，不过现在确实已经有很多破解脚本很乐意把你的程序运行上千次，程序中一旦有疏忽就会被它们抓住机会。一个聪明的破解脚本可能甚至会降低程序的优先级，使它比正常情况中断次数更多，这只是为了加快破解的速度。人们在这些方面很努力，这也是把它称为利用（exploits）的原因。

通过对已经打开的文件句柄调用`stat`，我们只访问一次文件名，这样就能避免竞态条件。为避免两个事件之间的竞争，一个好的策略是以某种方式将两个事件合并为一个，使这个操作成为一个原子操作<sup>注9</sup>。由于我们只按名访问一次文件，所以多次访问之间不存在竞态条件，名字是否改变并没有影响。即使黑客删除了我们打开的文件（没错，这也是有可能的），并换上了另外一个文件来诱骗我们，我们访问的仍是原来那个文件的句柄。

## 临时文件

除了允许缓冲区溢出（Perl脚本几乎不存在这个问题）以及信任不可信的输入数据（可以利用污染模式来防护），不适当地创建临时文件是最常被利用的安全漏洞。幸运的是，临时文件攻击通常要求黑客在他们想要攻击的系统上有一个合法的用户帐户，而这大大减少了可能的恶意黑客的数目。

---

注9：是的，在无核区也可以完成原子操作。德谟克利特把这种不可分的物质起名为“原子”时，他的意思是这种东西不能切割：ἀ-（不）+ τ ο μ ο ς（可切割）。原子操作是指不可中断的动作（就好像试图中断一个原子弹爆炸一样）。

不谨慎或很随便的程序可能会用各种不安全的方式使用临时文件，如把临时文件放在公共可写的目录中，使用可预测的文件名，以及没有首先确认文件是否并不存在。只要看到有以下代码的程序：

```
open(TMP, "> /tmp/foo.$$")
|| die "can't open /tmp/foo.$$: $!";
```

你会立即发现以上所有的这3种错误。这个程序肯定会出问题。

要利用这些漏洞，黑客首先会放入一个与你所用文件同名的文件。追加PID并不足以保证唯一性，尽管听起来让人很惊讶，不过要知道猜出PID并不难<sup>注10</sup>。程序中接下来就是一个不小心的open调用，这里并不是完成它本来的目的：创建一个新临时文件，实际上它会覆盖黑客的文件。

这有什么危险呢？危险有很多。要知道，黑客的文件实际上并不是一个普通文件。这是一个符号链接（或者有时是一个硬链接），可能指向黑客自己通常无法写的某个重要文件，如/etc/passwd。程序以为它打开了/tmp中的一个全新的文件，但实际上会删除别处的一个已有的文件。

Perl提供了两个函数来解决这个问题（如果使用得当）。第一个函数是POSIX::tmpnam，它会返回一个文件名（你可以安全地打开这个文件名）：

```
# 不断尝试文件名，直到得到一个全新的文件名
use POSIX;
do {
    $name = tmpnam();
} until sysopen(TMP, $name, O_RDWR | O_CREAT | O_EXCL, 0600);
# 现在使用TMP句柄完成I/O
```

第二个函数是IO::File::new\_tmpfile，它会返回一个已打开的句柄：

```
# 否则让模块为我们处理
use IO::File;
my $fh = IO::File::new_tmpfile(); # 这是POSIX的tmpfile(3)
# 现在使用$fh句柄完成I/O
```

这两种方法都不完美，不过在这两个方法中，第一个更好一些。第二个方法的主要问题是，Perl可能会受你的系统中C库的tmpfile(3)实现的影响，不能保证这个函数不会做危险的事情（就像我们正在努力修正的open）。而且，非常遗憾的是，一些实现中这个函数确实会做危险的事情。另外还有一个小问题，它根本不提供文件名。如果能处理临时文件而不需要文件名，这样当然更好，因为这样就不会由于再次打开文件而引入一个竞态条件，但通常情况下无法做到不用文件名。

第一种方法的主要问题是，不能像使用C库的mkstemp(3)函数那样控制路径名的位置。一

---

注10：除非你使用类似OpenBSD的系统，它的新PID是随机赋值的。



方面，你不希望把文件放在一个NFS装载的文件系统上。不能保证O\_EXCL标志在NFS下总能正确工作，所以如果有多个进程几乎同一时间请求一个排他创建（create），它们可能都会成功。另一方面，由于返回的路径可能在其他人可写的一个目录中，有人可能会放置一个符号链接，指向一个不存在的文件，强制你在他指定的位置创建你的文件<sup>注11</sup>。如果你要在文件中写内容，不要把临时文件放在其他人可写的目录中。如果必须这么做，调用sysopen时一定要使用O\_EXCL标志，另外务必使用已设置“仅所有者能删除”标志（粘贴位）的目录。

对于v5.6.1，还有第3种方法。标准File::Temp模块考虑了我们提到的所有困难。你可以使用默认选项，如：

```
use File::Temp "tempfile";
$handle = tempfile();
```

或者可以指定其中一些选项，如下所示：

```
use File::Temp "tempfile";
($handle, $filename) = tempfile("plughXXXXXX",
                                DIR => "/var/spool/adventure",
                                SUFFIX = ".dat");
```

File::Temp模块还为我们提到的其他函数提供了模拟函数，不过这些函数考虑到了上述安全问题，有更好的接口，因为它们会提供一个打开的文件句柄，而不是文件名（文件名存在竞态条件的风险）。要了解有关的选项和语义，更详细的描述请参见这个模块的文档。

一旦得到文件句柄，可以用它做你想做的任何处理。文件句柄可以读写，所以你可以写至句柄，再使用seek回到起始位置，如果你愿意，还可以覆盖刚才放在那里的内容，或者再读回那些内容。你真正要避免的不是再次打开文件，而是再次打开那个文件名，因为无法确定它是否确实是你第一次打开的那个文件<sup>注12</sup>。

从脚本启动另一个程序时，正常情况下Perl会为你关闭所有文件句柄，以避免另一个脆弱性。如果使用fcntl清除close-on-exec（执行时关闭）标志（如第27章介绍open时最后的例子所示），你调用的其他程序会继承这个新的打开的文件描述符。在支持/dev/fd/目录的系统上，可以向另一个程序提供实际上表示文件描述符的文件名，如下所示：

```
$virtname = "/dev/fd/" . fileno(TMP);
```

如果要调用的Perl子例程或程序只想要一个文件名作为参数，而且你知道这个子例程或程

---

注11：这个问题的解决方法是（只适用于某些操作系统）：调用sysopen并以OR方式结合O\_NOFOLLOW标志。如果路径的最后一部分是一个符号链接，函数调用会失败。

注12：除非以后对两个文件句柄都执行stat，并比较它们的前两个返回值（device/inode）。不过那时已经为时过晚，因为破坏已经造成了。你能做的只是检测到这个破坏，并退出（可能还会悄悄地向系统管理员发封email）。



序使用常规的open打开文件，那么可以使用Perl的文件句柄表示法来传递一个句柄：

```
$virtname = "=&" . fileno(TMP);
```

将文件“名”传入常规的单参数或两参数Perl open时（但不能是三参数open，否则将失去这个魔法），就能访问这个复制的描述符。在某些方面，这比从/dev/fd/传递一个文件可移植性更好，因为只要能用Perl的地方就可以使用这个方法；而不是所有系统都有一个/dev/fd/目录。另一方面，这种特殊的Perl open语法（按数字访问文件描述符）只适用于Perl程序，而不能用于其他语言编写的程序。

## 处理不安全的代码

如果你想捕获本该由你自己捕获的虚假数据，但是在数据真正转移到系统之前不想贸然捕获，在这种情况下，污染检查就是你需要的那种安全毯。这有点像Perl提供的那些可选的警告。它们可能并不指示一个真正的问题，不过一般来讲，处理误报的错误与未能处理漏报的错误比起来，前者痛苦要轻得多。由于存在污染，后者的痛苦更持久，因为使用虚假数据不只会给出错误的答案。还可能将你的系统彻底摧毁，连同你最近两年的工作都随之灰飞烟灭（如果你没有很好地备份，甚至还可能包括你接下来两年的工作）。也许你相信自己写的代码是可信的，但是不能肯定向你提供数据的其他人不会哄骗你做出让你后悔的事情，这种情况下污染模式就很有用。

数据是一方面。如果你甚至不相信自己运行的代码，那又是另一回事。如果你从网络上得到一个applet，其中包含一个病毒、时间炸弹或者特洛伊木马，那该怎么办？在这里污染检查是没有用的，因为你输入程序的数据没有问题——不可信的是代码。你把自己置于一个特定的位置上，就好像从一个陌生人那里收到一个神秘的设备，并附有一张字条告诉你“把它对着你的脑袋，然后扣动扳机”。也许你会认为这个设备可以用来吹头发，不过很快你就不会那么想了（也许你已经没有机会想了，因为它可能是一把手枪）。

在这个领域，谨慎与偏执是同义词。你想要的是一个允许你隔离可疑代码的系统。代码可以继续存在，甚至可以完成某些功能，但是你不允许它到处漫游伺机想做什么就做什么。在Perl中，可以使用Safe模块来强制隔离。

## 改变Root

Perl中chroot的工作类似于chroot(2)系统调用。它可以改变root目录，从而使程序不能访问你打算使用的那部分文件系统之外的任何文件。不过，既然只有root用户可以使用这个方法，所以这里已经存在一个安全问题：

```
chroot( '/usr/local/apache/data' );  
chdir( '/' ); # 现在位于/usr/local/apache/data
```

实际上这并不能阻止访问新root目录以外的文件。尽管不能使用文件名，但使用chroot之前已经打开的目录句柄可能会偷偷指向真正的root（'/'）：

```
use v5.14;
use warnings;

say "Here I am";

opendir my $rootdh, '/';

chroot( '/Users/Amelia' );
opendir my $dh, '/'; # /Users/Amelia
say for readdir($dh);

chdir( $rootdh ); # 唉呀，又回到真正的 '/'
opendir my $dh, '.';
say for readdir($dh);
```

发生这种情况的前提是你或其他人允许它发生。如果有人编辑程序来插入这种捣乱的代码，任何级别的Perl安全都无济于事。一定要用你能找到的所有手段避免有root用户权限的Perl程序。

## 安全隔离室

Safe模块允许你建立一个沙箱（sandbox），这是一个特殊的隔离室，可以在其中捕获所有系统操作，而且命名空间访问也会得到小心地控制。这个模块的底层技术细节还在一直变动中，所以这里更侧重介绍相关的理论。

### 限制命名空间访问

在最基本的层次上，Safe对象就像一个保险箱，只不过它的想法是把坏人关在里面，而不是关在外面。在Unix世界中，有一个名为chroot(2)的系统调用，可以永久地限制一个进程只能在目录结构的一个子目录中运行（如果你愿意，可以称之作它自己的小世界）。一旦把进程放在那里，它就没有办法再触及外面的文件，因为它无法命名外面的文件<sup>注13</sup>。

Safe对象有点像chroot(2)，只不过并非限制在文件系统目录结构的一个子集中，而是限制在Perl包结构的一个子集中，Perl包结构与文件系统一样也是层次结构。

还可以采用另外一种方法来理解Safe对象，可以把它看作是那种有单向镜子的观察室，警察把嫌疑人关在这种观察室里。外面的人可以看到屋里的情况，但是里面的人看不到外面。

---

注13：有些网站会这样做，可能使用只读方式加载的环回文件系统来执行所有CGI脚本。这会很麻烦，不过，即使有人逃脱，也会发现无处可逃。



创建一个Safe对象时，如果愿意，可以为它指定一个包名。如果没有指定包名，则会为你选择一个新包名：

```
use Safe;
my $sandbox = Safe->new("Dungeon");
$Dungeon::foo = 1; # 不过，不允许直接访问
```

如果使用提供给new方法的这个包名来完全限定变量和函数，则可以从外面访问这个包中的这些变量和函数，至少在当前实现中是这样。

稍稍更向上兼容的做法是，在创建Safe之前先完成设置，如下所示。这种做法仍然有效，如果要建立有很多初始“状态”的Safe，这是一种很方便的做法（当然必须承认，\$Dungeon::foo没有太多状态）。

```
use Safe;
$Dungeon::master = 'Gary Gygax'; # 仍是直接访问，仍然不建议这么做
my $sandbox = Safe->new("Dungeon");
```

不过，Safe还提供了一种访问隔离室全局变量的方法（即使你不知道隔离室包的名字）。所以为了得到最大的向上兼容性（不过速度最慢），建议你使用reval方法：

```
use Safe;
my $sandbox = Safe->new();
$sandbox->reval( q($master = 'Gary Gygax') );
```

实际上，运行可疑代码时使用的也是这个方法。向隔离室传入代码进行编译和运行时，这个代码会认为自己实际上在main包中。外部世界调用\$Dungeon::master时，内部代码会认为它是\$main::master、\$::master或者就是\$master（如果不是在use strict下运行）。在这个隔离室内部调用\$Dungeon::master是不行的，因为这实际上会访问\$Dungeon::Dungeon::master。通过让Safe对象有自己的main概念，程序中其余的变量和子例程就能得到保护。

要在这个隔离室中编译和运行代码，可以使用reval（“受限eval”）方法，将代码字符串作为参数传入。与所有其他eval STRING 构造类似，reval中如果出现编译错误和运行时异常并不会终止你的程序，只会退出reval，将异常保留在\$@中，所以每个reval调用之后都要检查\$@。

使用前面给出的初始代码，它会打印“master is now Dave Arneson”，但前提是你允许使用print（见下一节）：

```
$sandbox->permit( qw(print) );
$sandbox->reval(
    q($master = 'Dave Arneson'; print "master is now $main::master\n");
);
if ($@) {
    die "Couldn't compile code in box: $@";
}
```



```
}
```

如果你只想编译代码，但不运行，可以把代码字符串包装在一个子例程声明中：

```
$sandbox->reval(q{
    our $master;
    sub say_master {
        print "master is now $main::master\n";
    }
}, 1);
die if $@;      # 检查编译
```

这一次我们向reval传入了第二个参数，由于它为true，这会告诉reval要在strict pragma下编译代码。从代码字符串无法禁用严格性检查，因为通常不能在Safe隔离室中完成导入和取消导入（在Safe隔离室中通常有很多事情都不能做—参见下一节）。

一旦在隔离室中创建了say\_master函数，下面的方法基本上是一样的：

```
$sandbox->reval("say_master()");      # 最好的方法
die if $@;

$sandbox->varglob("say_master")->();  # 通过匿名类型团调用

Dungeon::say_master();                # 直接调用，强烈建议不要使用这种方法
```

## 限制操作符访问

关于Safe对象还有很重要的一点，Perl限制了沙箱中可用的操作（你可能允许你的孩子拿着小桶和小铲在沙箱里玩，但是可能不会愿意让他拿着火箭筒玩）。只是保护程序的其余部分还不够，你还要保护计算机的其余部分。

在一个Safe对象中编译Perl代码时，可以用reval或rdo（受限的do *FILE*操作符），编译器会查看针对每个隔离室的一个特殊的访问控制列表，来确定各个操作是否可以安全地编译。这样一来，你就不用过于担心那些不可预见的shell转义、意外地打开文件、正则表达式中奇怪的代码断言，或者通常让人不安的（或应当考虑的）大多数外部访问问题。

如果你想改变拒绝或允许的操作，可以告诉隔离室要限制或允许哪些操作：

```
use v5.10;

$time = $sandbox->reval( q(time) );    # 正常工作

$sandbox->deny( qw(time) );
$time = $sandbox->reval( q(time) );    # 失败
```

可以限制Opcode模块中指定的整个操作码集合（见表20-1），不过这要求对Perl的内部原理有所了解：

```
$sandbox->deny( qw(:base_math) );
my $time = $sandbox->reval( 'log(10)' );    # 失败
```

表20-1: Opcode中的部分操作码标记

操作码	包含
:base_io	基于文件名的输入和输出
:dangerous	各种危险事物的垃圾场标记
:filesystem	输入和输出
:load	加载外部文件或得到调用者信息
:sys_db	访问系统数据库, 如/etc/passwd
:subprocess	创建子进程

不过, 这是在保护Opcode, 确保它的导出标记确实是你期望的。如果你不信任这些标记, 也可以指定单个操作码 (也参见Opcode模块的文档)。不要相信任何人。

Safe模块没有提供对拒绝服务攻击 (denial-of-service attacks) 的完全保护, 特别是采用权限比较宽容的模式时。拒绝服务攻击会消耗某种类型的全部可用系统资源, 拒绝其他进程访问基本的系统设施。这种攻击的例子包括填满内核进程表、在紧凑循环中无休止地运行从而完全占用CPU、耗尽可用内存, 以及填满文件系统。这些问题很难解决, 特别是很难以可移植的方式解决。参见本章后面“代码伪装为数据”一节, 其中对拒绝服务攻击有更深入的讨论。

## 安全示例

假设有一个CGI程序要管理一个表单, 用户可以在这个表单中输入任意的Perl表达式, 然后得到计算的结果<sup>注14</sup>。与所有外部输入一样, 输入的字符串是污染的, 所以Perl不会允许你用eval计算它。首先必须用一个模式匹配清洗这个输入。问题在于, 你可能无法设计出一个能够检测出所有可能威胁的模式。而且你也不敢对这些输入完成清洗之后就把它发送给内置的eval (如果你这么做, 我们会进入你的系统, 然后删除这个脚本)。

这里就要用到reval。下面的CGI脚本会处理一个表单 (其中只有一个表单域), 这个脚本 (在标量上下文中) 计算由表单域得到的字符串, 然后打印格式化的结果:

```
#!/usr/bin/perl -lTw
use strict;
use CGI::Carp "fatalsToBrowser";
use CGI qw/:standard escapeHTML/;
use Safe;

print header(-type => "text/html;charset=UTF-8"),
      start_html("Perl Expression Results");
my $expr = param("EXPR") =~ /^([^;]+)/
    ? $1 # 返回现在无污染的部分
```

注14: 不要笑。我们确实见过有Web页面这么做, 而且没有用Safe!

```

        : croak("no valid EXPR field in form");
my $answer = Safe->new->reval($expr);
die if $@;

print p("Result of", tt(escapeHTML($expr)),
        "is", tt(escapeHTML($answer)));

```

假设有恶意用户输入“`print `cat /etc/passwd``”（或者更糟）作为输入字符串。由于这个受限环境中不允许使用反引号，Perl会在编译期间捕获这个问题并立即返回。`$@`中的字符串是“quoted execution (`` , qx) trapped by operation mask”，另外后面还有一些定制信息指出问题出现在哪里。

由于没有另行说明，我们创建的隔离室都使用默认的允许操作集。至于如何将特定的操作声明为允许或禁止，在这里并不重要。重要的是这些完全在你的程序的控制之下。另外，由于可以在程序中创建多个Safe对象，可以根据代码的来源对不同的代码块赋予不同的信任度。

如果你想利用Safe，下面给出一个交互式的Perl小计算器。可以在这个计算器中输入数值表达式，你会立即看到表达式的计算结果。不过这并不仅限于数字。它更像是第27章中介绍eval时给出的循环例子，可以接收用户提供的任何输入、完成计算，然后返回结果。不同之处在于Safe版本不会执行你提交的任何代码。可以在终端上交互式地运行这个计算器，输入一点Perl代码并查看答案，感受一下Safe能提供哪些保护。

```

#!/usr/bin/perl -w
# safecalc - 使用Safe的演示程序
use strict;
use Safe;
my $sandbox = Safe->new();
while (1) {
    print "Input: ";
    my $expr = <STDIN>;
    exit unless defined $expr;
    chomp($expr);
    print "$expr produces ";
    local $SIG{__WARN__} = sub { die @_ };
    my $result = $sandbox->reval($expr, 1);
    if ($@ =~ s/at \ (eval \d+\) .*//) {
        printf "[%s]: %s", $@ =~
            /trapped by operation mask/
            ? "Security Violation" : "Exception", $@;
    }
    else {
        print "[Normal Result] $result\n";
    }
}

```

如果提供正常的代数表达式，它会完成计算，并返回结果。如果你有非法企图，例如试图运行反引号或者加载一个模块，它就会“严词”拒绝：



```
Input: 2+2
2+2 produces [Normal Result] 4
Input: `ls -l`
`ls -l` produces [Security Violation]: 'quoted execution (``, qx)'
trapped by operation mask
Input: use LWP::Simple; getprint('http://www.perl.org')
use LWP::Simple; getprint('http://www.perl.org') produces [Security Violation]:
'require' trapped by operation mask
Input: 1/137
1/137 produces [Normal Result] 0.0072992700729927
```

## 代码伪装为数据

可以使用Safe隔离室防止可怕的事情发生，不过这并不意味着在家附近完成一些日常活动时完全可以放松警惕。你要养成一种习惯，对你的周围环境始终抱有警惕的态度，要从那些试图入侵者的角度考虑问题。你要采取一些主动预防措施，如保证光照，而且要及时修剪门前的灌木，很多潜伏作案的人都会藏在那些隐蔽的角落。

在这个领域Perl也会帮助你。Perl传统的解析和执行机制可以避免shell编程语言常常落入的一些陷阱。Perl语言中有很多极其强大的特性，不过根据设计，它们在语法和语义上都会有所限制，保证一切都在程序员的控制之下。除了很少的几个例外，一般情况下Perl中各个token只计算一次。如果一个东西看起来要用作简单数据变量，那么它就是简单数据变量，不会突然变成代码肆意“翻弄”你的文件系统。

遗憾的是，如果你调用shell为你运行其他程序，则有可能发生这种事情，因为这样一来你是在shell的规则下运行，而不是在Perl的规则下运行。不过，很容易避免使用shell，只需要使用列表参数形式的system、exec或管道式open函数。虽然反引号没有列表参数形式来防范shell，不过完全可以模拟，如本章前面“降权限访问命令和文件”一节所述（尽管在语法方面没有办法让反引号接受一个参数列表，但我们正在开发一个多参数形式的底层readpipe操作符，不过写这本书时它还不够成熟）。

在一个表达式中使用变量时（包括将变量内插到一个双引号字符串），变量没有机会包含Perl代码来做你不打算做的事情<sup>注15</sup>。与shell不同，Perl不需要对变量加引号保护（不论变量中是什么）。

```
$new = $old;           # 不需要加引号
print "$new items\n";   # $new不会有危害

$phrase = "$new items\n"; # 这里也不会
print $phrase;          # 这也没问题
```

Perl采用一种“所见即所得”的方法。如果你没有看到另外一层内插，那么内插就不会

---

注15：不过，如果要生成一个Web页面，可能会发出HTML标记，包括JavaScript代码，这可能会做远程浏览器不想做的事情。

发生。也有可能把任意的Perl表达式内插到字符串中，不过前提是你特别要求Perl这么做（即便如此，如果在污染模式中，还会对这些内容进行污染检查）。

```
$phrase = "You lost @ {[ 1 + int rand(6) ]} hit points\n";
```

不过，内插不是递归的。不能在字符串中隐藏任意的表达式：

```
$count = "1 + int rand(6)";          # 一些随机代码
$saying = "$count hit points";        # 只是一个直接量
$saying = "@{[$count]} hit points";   # 也是一个直接量
```

`$saying`的两个赋值都会生成“1 + int rand(6) hit points”，而没有将`$count`的内插内容作为代码来计算。要让Perl这么做，必须显式地调用`eval STRING`：

```
$code = "1 + int rand(6)";
$die_roll = eval $code;
die if $@;
```

如果`$code`被污染，`eval STRING`会生成异常。当然，一般情况下你不会希望计算随机的用户代码，不过如果你真的想这样，应该考虑使用Safe模块。你可能已经听说过了。

在另外一种情况下，有时Perl会把数据处理为代码；具体来说，当`qr//`、`m//`或`s///`操作符中的模式包含新的正则表达式断言（`{ CODE }`）或（`??{ CODE }`）时，会作为代码处理。在模式匹配中，如果只是用作为直接量，它们不会带来安全问题：

```
$cnt = $n = 0;
while ($data =~ /( \d+ ({ $n++ }) | \w+ )/gx) {
    $cnt++;
}
print "Got $cnt words, $n of which were digits.\n";
```

不过如果现有代码中将变量内插到匹配，编写这些代码时总有一个假设，认为数据就是数据，而不是代码。因此，这个新构造可能会在原本安全的程序中引入安全漏洞。所以如果内插字符串包含一个代码断言，Perl会拒绝计算这个模式，并产生一个异常。如果你确实需要这个功能，可以用词法作用域声明`use re 'eval' pragma`来启用（不过，还是不能将污染数据用于内插代码断言）。

正则表达式还有一类完全不同的安全问题，这就是拒绝服务问题。这些问题可能让你的程序过早退出，运行太长时间，或者耗尽所有可用内存，有时甚至会转储内核，这取决于具体情况。

处理用户提供的模式时，不用担心解释随机的Perl代码。不过，正则表达式引擎有自己的小编译器和解释器，用户提供的模式会给这个正则表达式编译器带来麻烦。如果一个内插模式不是合法的模式，则会产生一个运行时异常，除非这个异常被捕获，否则它将成为致命错误。如果你尝试捕获这个异常，要确保只使用`eval BLOCK`，而不是`eval STRING`，因为后者会增加一层计算，这实际上会允许执行随机的Perl代码。应该这样做：

```
if (not eval { "" =~ /$match/; 1 }) {  
    # (现在对坏模式做你想做的任何处理)  
}  
else {  
    # 我们知道模式至少可以安全地编译  
    if ($data =~ /$match/) { ... }  
}
```

还有一个更麻烦的拒绝服务问题：尽管数据和搜索模式都正确，但程序看起来好像永远挂起了。这是因为有些模式匹配需要大量时间来计算，甚至很容易超出我们太阳系的平均无故障时间（MTBF）。如果你特别“幸运”，这些计算量极大的模式还需要极大的存储空间。如果是这样，你的程序就会耗尽所有可用的虚拟内存，使系统的其余程序统统停顿，让用户大为光火，要么带着规范的“Out of memory!”（内存耗尽）错误终止，要么留下一个极其庞大的转储文件（不过可能没有太阳系那么大）。

与大多数拒绝服务攻击类似，这个问题并不容易解决。如果你的平台支持alarm函数，可以为模式匹配限时。遗憾的是，Perl（目前）不能保证处理信号的单纯行为不会触发内核转储（这个问题计划在将来的版本中修正）。不过，总是可以尝试一下，另外即使信号没有得到妥善的处理，至少程序不会无休止地运行下去。

如果你的系统支持对各个进程施加资源限制，那么在调用Perl程序之前可以在你的shell中设置这些限制，或者使用CPAN的BSD::Resource模块直接从Perl设置。Apache Web服务器允许对它启动的CGI脚本设置时间、资源和文件大小限制。

最后，希望你能有这样一种感觉，觉得安全问题还没有完全解决。记住，即使你很谨慎，但这并不意味着不会遇到安全问题。所以不妨好好享受吧。



# 常用实践

几乎所有Perl程序员都很乐于告诉你一大堆关于如何编程的建议。我们也不例外（特别强调一下，以免你没有注意到）。在这一章中，我们不打算告诉你Perl的特殊特性，我们会换一个角度，使用一种类似霰弹枪的方法介绍Perl的惯用法。通过把这些看起来没有关联的东西汇合在一起，希望能让你对如何“以Perl方式思考”有些感觉。毕竟，你编程时并不是先写一堆表达式，再写一堆子例程，然后再写一堆对象。你必须或多或少同时照顾到所有这些方面。所以这一章的组织也有点这种意思。

不过，这一章确实也有一个基本的组织结构，我们先从负面建议开始，逐步过渡到正面建议。不知道这样会不会让你感觉好一些，不过这确实让我们感觉好多了。而且，在你的大部分编程生涯中，都要先花时间学习哪些不能做，然后再确定哪些要做，所以尽早熟悉这种先负面再正面的方式吧。

## 新手的常见失误

在所有这些失误中，最大的失误就是忘记使用`use warnings`，要知道使用`use warnings`能发现很多错误。另一个大失误是忘记在适当的时候使用`use strict`。当程序开始变得越来越大时（肯定会这样），这两个`pragma`可以让你省去很多烦恼。还有一个失误是忘记查看在线`perlfaq`。假设你想知道Perl是否有一个`round`函数，可以试着先搜索`perldoc`来查找FAQ：

```
% perldoc -q round
```

除了这些“高层次的失误”，还有很多编程陷阱。有些陷阱几乎每个人都掉进过，另外一些陷阱只会让有不同文化背景的人掉进去。我们会在下面的小节中区分这两种陷阱。

## 常见错误

- `print`语句中在文件句柄后面加逗号。尽管看起来这样写很正常：

```
print STDOUT, "goodbye", $adj, "world!\n";          # 不正确
```

但由于多了第一个逗号，这是不正确的。你真正想要的是间接对象语法：

```
print STDOUT "goodbye", $adj, "world!\n";          # 正确
```

这里的语法是正确的，所以你可以写以下代码：

```
print $filehandle "goodbye", $adj, "world!\n";
```

这里`$filehandle`是一个标量，包含运行时一个文件句柄的名字。

这与下面的代码不同：

```
print $notafilehandle, "goodbye", $adj, "world!\n";
```

其中`$notafilehandle`只是一个字符串，它是要打印的结果列表的一部分。对于这种情况，你可能会在终端上看到类似`GLOB(0xDEADBEEF)`的输出，因为输出结果将进入标准输出（`STDOUT`），所以文件句柄引用会自行字符串化。

参见术语表中的间接对象（indirect object）。

- 使用`==`而不是`eq`，另外使用`!=`而不是`ne`。`==`和`!=`操作符都是数值测试。另外两个则是字符串测试。字符串`"123"`和`"123.00"`作为数字是相等的，但作为字符串则并不相等。另外，大多数非数值字符串在数值上都等于0，而且其中一些（如`"123xyz"`）在数值上下文中可能与你所想的不一樣。除非在处理数字，否则几乎总会用字符串比较操作符。如果这些操作符使用了非数值数据，`warnings pragma`会指出这一点。
- 忘记末尾的分号。Perl中的每一条语句都以一个分号或块末尾来结束。换行符并不是语句结束符，这与`awk`、`Python`或`FORTRAN`中不同。要记住，在这方面Perl与C类似。

包含here文档的语句尤其容易漏掉分号。正确的写法如下：

```
print <<"FINIS";
A foolish consistency is the hobgoblin of little minds,
adored by little statesmen and philosophers and divines.
                                --Ralph Waldo Emerson

FINIS
```

- 忘记块（`BLOCK`）需要大括号。裸语句不是块。如果你要创建一个控制结构（如`while`或`if`），需要多个块，就必须在每个块周围使用大括号。要记住，在这方面Perl与C不同。
- 没能在正则表达式之间保存`$1`、`$2`等。记住，每个成功的`m/match/`（匹配）或`s/ubstitution/`（替换）都会设置（清除或修改）`$1`、`$2`……变量。要想立即保存这些变量，一种方法是在列表上下文中计算匹配，如下：



```
my ($one, $two) = /(\w+) (\w+)/;
```

- 对于一个`local`变量，由于它是用`local`声明的，其作用域中调用的其他子例程所看到的变量值会改变，但我们往往没有意识到这一点。因为在类似C的语言中没有与之对应的构造，所以很容易忘记`local`是一个运行时语句，它会设置动态作用域。参见第4章“作用域声明”一节。通常可能应当用`my`来声明变量。
- 大括号未正确配对。一个好的文本编辑器会帮助你匹配大括号。请找一个（或两个）这样的编辑器。它还有助于建立一种一致的风格，使你知道哪里会有一个大括号（尽管有人对此可能有异议，认为这些位置另有用处）。`Perl::Tidy`之类的工具可以为你美化代码。
- 在`do {} while`中使用循环控制语句。尽管这个控制结构中的大括号看起来有点像循环块的一部分，但实际上它们并不是。
- 本意是使用`$foo[0]`，但使用了`$foo[1]`。Perl数组默认地从0开始索引。原先Perl曾经想提供灵活性，允许你通过`$[`特殊变量设置起始索引，不过v5.12废弃了这种做法。
- 本意是使用`$foo[0]`，但使用了`@foo[0]`。`@foo[0]`引用是一个数组片段，表示由单个元素`$foo[0]`组成的一个数组。有时这并没有差别，如：

```
print "the answer is @foo[0]\n";
```

但对于以下情况，差别就很大了：

```
@foo[0] = <STDIN>;
```

这会收入STDIN其余的所有内容，将第一行赋给`$foo[1]`，并丢掉所有其他行。这可能不是你想要的。要养成习惯，认为`$`表示单个值，而`@`表示一个值列表，如果能做到这一点，你就不会有问题。

- 忘记类似`my`等列表操作符要加小括号，这会使一个变量作为词法作用域变量，而其他变量为全局变量：

```
my $x, $y = (4, 8); # 不正确
my ($x, $y) = (4, 8); # 正确
```

- 在设置格式变量`$^`、`$~`或缓冲变量`$|`之前忘记选择适当的文件句柄。这些变量依赖于当前选择的文件句柄（由`select(FILEHANDLE)`确定）。初始选择的文件句柄为STDOUT。应当使用`IO::Handle`模块的文件句柄方法。参见第25章。
- 忘记对读写的每个文本流设置编码。没有通用“文本文件”之类的东西。为了方便，`-C`命令行选项、`PERL_UNICODE`环境变量，以及`open pragma`可以隐式设置编码；为了保证准确，`binmode`和`open`函数可以显式设置编码。如果没有以某种方式隐式或显式地指定编码，你就得不到文本数据。编码是不能猜的，必须明确指定。



## 常被忽略的建议

Perl程序员应该把这些记下来：

- 要记住，很多操作在列表上下文和标量上下文中会有不同的表现，或者列表和数组是不同的。例如：

```
($x) = (4, 5, 6); # 列表上下文; $x设置为4
$x = (4, 5, 6); # 标量上下文; $x设置为6

@a = (4, 5, 6);
$x = @a; # 标量上下文; $x设置为3 (数组长度)
```

- 尽可能避免裸字，尤其是全小写的裸字。只从字面上看，你无法区分它是一个函数还是一个裸字字符串。如果对字符串加引号，另外对函数调用参数两边加小括号，就不会把它们弄混了。实际上，程序最前面的`pragma use strict`会使裸字成为一个编译时错误，这可能是一件好事。
- 如果只从字面上看，无法区分哪些内置函数是一元操作符（如`chop`和`chdir`），而哪些是列表操作符（如`print`和`unlink`），另外哪些是无参数的操作符（如`time`）。可以在第27章更多地了解这些内置函数。一般来讲，如果不确定，或者即使你不确定自己是否确定，那么就使用小括号。另外，用户自定义的子例程默认都是列表操作符，不过可以用一个原型（`$`）声明为一元操作符，或者用原型（`()`）声明为无参数操作符。
- 有些函数会默认处理`$_`、`@ARGV`或其他参数，而另外一些没有默认参数，人们很难记住这一点。要花些时间来学习哪些有默认参数，或者如何避免这些默认参数。
- `<FH>`不是一个文件句柄的名字。它是一个尖角操作符，会对句柄完成一个行输入操作。如果想要打印输出到这个尖角操作符，就会出现混乱：

```
print <FH> "hi";          # 不正确，忽略尖角
```

- 还要记住，在一个`while`循环中，只有当读文件是唯一的条件时，尖角操作符读取的数据才会赋给`$_`：

```
while (<FH>) { }          # 数据赋给$_
<FH>;                    # 读取数据并丢掉
```

- 需要`=~`时不要使用`=`，这两个构造有很大区别：

```
$x = /foo/;              # 在$_中搜索"foo"，将结果放入$x
$x =~ /foo/;             # 在$x中搜索"foo"，丢弃结果
```

- 在替换中使用`/r`返回结果。

```
@new = map { s/old/new/r } @old;
```

- 尽可能使用`my`声明局部变量。使用`local`只会为全局变量提供一个临时值，这会使得你遭遇动态作用域的很多不可预见的副作用。

- 不要对模块的导出变量使用`local`。如果局部化一个导出的变量，导出的变量值不会改变。局部名会成为一个新值的别名，而外部名仍是原变量的别名。

## C陷阱

有心的C程序员应该把这些记下来：

- `if`和`while`块要有大括号。
- 必须使用`elsif`而不是“`else if`”或“`elif`”。下面的语法是不合法的：

```
if (expression) {
    block;
}
else if (another_expression) { # 不正确
    another_block;
}
```

`else`部分总是一个块，而裸`if`不是一个块。不能期望Perl与C完全一样。应该使用以下语法：

```
if (expression) {
    block;
}
elsif (another_expression) {
    another_block;
}
```

还要注意“`elif`”就是“`file`”倒过来拼。只有Algol程序员喜欢这样：一个关键字倒着拼成为另一个关键字。

- C的`break`和`continue`关键字在Perl中分别变成`last`和`next`。与C中不同，它们不能在`do {} while`构造中使用。
- Perl在很长一段时间里都没有提供与C中`switch`语句等价的构造。Perl v5.10引入了一个“增强型`switch`”，它有一个有趣的名字`given-when`（因为它本身就是一个有趣的构造）。参见第4章。建立你自己的“`switch`”构造也很容易，参见第4章中的“裸块作为循环”和“`given`语句”。
- Perl中变量以`$`、`@`或`%`开头。
- 注释以`#`开头，而不是`/*`。多行注释要用`pod`。
- 不能取地址，不过Perl中有一个类似的操作符，即反斜线，它会创建一个引用。如果对一个引用字符串化，你会得到一个类似地址的结果，不过不能用它做任何事情。
- `ARGV`必须大写。`$ARGV[0]`是C中的`argv[1]`，C的`argv[0]`在Perl中是`$0`。参见第25章。
- `link`、`unlink`和`rename`等系统调用成功时会返回`true`而不是`0`。

- %SIG中的信号处理器处理信号名，而不是数字。

## Shell陷阱

聪明的shell程序应该把这些记下来：

- 赋值左边以及右边的变量都要有\$、@或%前缀。如以下的“shell风格”赋值不会像你期望的那样解析：

```
camel="dromedary";          # 不正确
```

正确的赋值是：

```
$camel="dromedary";         # 正确
```

- foreach的循环变量总是需要一个\$。尽管csh中是这样：

```
foreach hump (one two)
    stuff_it $hump
end
```

但在Perl中要写为：

```
foreach $hump ("one", "two") {
    stuff_it($hump);
}
```

- 反引号操作符会完成变量内插，而不考虑命令中是否存在单引号。
- 反引号操作符对返回值不做任何转换。在Perl中，必须显式地去除换行符，如下所示：

```
chomp($thishost = `hostname`);
```

- Shell（尤其是csh）会对各个命令行完成多层替换。Perl只在类似双引号、反引号、尖括号和搜索模式等构造中完成内插。
- Shell一次只解释一点脚本。Perl会在执行程序之前编译整个程序（除BEGIN块以外，BEGIN块会在编译完成之前执行）。
- 程序参数可以通过@ARGV得到，而不是通过\$1，\$2等变量。
- 不能作为单个标量变量自动得到环境。如果希望得到环境，可以使用标准Env模块。

## Python陷阱

Perl和Python都是动态语言，它们有一些共同的祖先，而且在1987年到1991年这5年间，在它们身上彼此都能看到对方的身影。Perl 4甚至还从Python窃取了对象系统，并把它加入到Perl 5中。从这两个语言的语法就可以看出二者的相似性，不仅如此，从其他方面也可以看到很多相同之处。



- Python和Perl有时使用不同的词表示相同的概念，有时则使用相同的词表示不同的概念。参见表21-1。

表21-1：Python与Perl术语映射

Python	Perl
元组 (Tuple)	列表 (List)
列表 (List)	数组 (Array)
字典 (Dictionary)	散列 (Hash)

- Perl中变量以\$、@或%开头。通过使用类似`$str`的印记，Perl可以保证区分其名词和动词，这样你就不用担心无意中覆盖某些重要的内置函数，这与Python不同，在Python中，使用你自己的`str`时会覆盖Python的内置函数。在Perl中也可以覆盖内置函数，不过不会像Python中那样意外地覆盖。
- 不要忘记使用`use warnings`，这会让Perl注意到一些问题，而在Python中这些问题会作为异常指出。在Perl中，如果没有`use warnings`，除非你特别测试这些问题时才会知道问题的存在，所以如果你忘记使用`use warnings`声明，就永远也不会知道有这些问题。参见第27章的`warnings`和`autodie`。
- 很多内置函数有默认参数，或者对于最常见的情况有默认行为。参见第27章。
- Python的方法需要显式的参数表。在Perl中，要对函数的参数解包 (`unpack`)，这会为你提供很大的灵活性，可以自由指定函数参数的个数和顺序。我们把这看作是一个特性，不过如果你发现需要写大量样板来解包函数参数，那么可以考虑使用CPAN上的`Method::Signatures`模块。
- Perl很了解模式，编译时除了编译程序的其余部分外还会为你编译模式。
- Perl的`\N{NAME}`构造接受快捷方式、别名和定制名（甚至在不同的词法作用域可以不同）；Python的`\N{NAME}`则不接受。
- Perl字符是抽象码点，而Python中则是底层代码单元。
- Perl模式匹配使用Unicode规则来实现不区分大小写的比较，而Python只使用ASCII大小写转换规则，所以（例如）不区分大小写时，Perl中3个希腊字符`sigma`都能匹配。
- Perl的大小写映射函数（如`uc`和`lc`）遵循Unicode规则，所以它们应用于所有有大小写的码点，而不只是字母。
- Perl了解（可能嵌套的）词法作用域，所以可以坦然面对完全词法作用域闭包。而Python不了解，也不能很好地应对。
- Perl使用完全Unicode大小写，所以一个字符串的大小写映射可能比原来的字符串

长。Python只使用简单Unicode大小写（如果使用的话），不能为字符串提供很好的结果。

- 任何子例程如果返回被祝福的引用，它们就都是Perl构造函数。构造方法没有一个特殊的名字。
- Perl方法就是方法，它们总会接收其调用者作为一个额外的初始参数。作为一个语言，Perl并不按Python那样区分对象方法、类方法或静态方法。
- Perl的面向对象性是可选的，而不是强制的。Perl并不强制其内置类型都采用面向对象模式，除非你有这个要求（甚至并不是所有类型都有方法）。不过你可能会喜欢autobox。
- 在Perl中，可以如下调用一个有参数的函数：

```
my $string = join("|", qw(Python Perl Ruby) );
```

在Python中，往往有一个主参数，它有一个方法来完成处理：

```
new = "|".join(["Python", "Perl", "Ruby"])
```

- Perl的模式匹配会浮动，除非显式地锚定模式的位置，类似于Python的`re.search()`方法，但不同于它的`re.match()`，`re.match()`只能匹配一行的开始。
- Perl的字符串不是字符数组，所以不能对字符串使用数组操作符。在字符串上当然只能使用字符串操作符。
- 除了反斜线本身或反斜线定界符，Perl不会扩展单引号字符串中的反斜线转义，不过Python会这么做。Perl的单引号字符串（如`'\t'`）更像是Python的原始字符串（如`r'\t'`）。
- Perl使用反引号对直接量加引号，来执行任意的系统命令并返回其输出，如`$file = `cat foo.c``。
- 在Perl中不用像Python中那样重新分配内存，因为数组和其他数据结构会根据需要扩展，有时会通过自动生成（autovivification）来完成。在Python中，必须显式地扩展列表，另外必须显式地分配新列表和字典来完成扩展。
- Python会对`open`失败等正常操作抛出异常，而Perl会使用特殊的返回值，通常是`undef`。这说明，如果你忘记检查错误返回，就会漏掉这个错误。可以使用`autodie`让失败的系统调用产生异常。
- 如果字符串到数值的转换失败或部分失败，默认情况下Perl并不会为此抛出异常，另外如果将`undef`当作一个已定义的值，Perl也不会抛出异常。可以用以下方法让它抛出异常：

```
use warnings FATAL => q(numeric uninitialized);
```

- Perl列表不会嵌套（即使增加了额外的小括号）。要建立（数组引用的）嵌套数组，需要使用中括号。
- Perl的range操作符包含范围的上下界，所以0..9包含0和9。
- Perl的交互式shell是它的调试器（第17章），不过Devel::REPL也很不错。如果调用Perl时没有提供参数，并不会像Python中那样进入一个交互式的read-eval-print循环。要想进入这样一个循环，应当使用perl -de0。

## Ruby陷阱

Ruby的创始人Matz从Perl“借鉴”了很多东西（我们认为他确实选择了一个不错的起点）。实际上，他把Perl和Smalltalk关在一起，让它们繁衍得到了一个新的产物。

- 没有irb。参见Python部分。
- Perl只考虑数字，它不关心有没有小数部分。
- 不需要用{}包围变量（Ruby中要用#{})来完成内插，除非你需要区分标识符和它周围的字符串：

```
"My favorite language is $lang"
```

- Perl内插的字符串不用加双引号：可以使用qq指定任意的定界符。类似地，通用的非内插字符串不必使用单引号：它们可以使用q指定任意的定界符。

```
q/That's all, folks/
q(No interpolation for $100)
qq(Interpolation for $animal)
```

- 所有Perl语句要用;分隔，即使它们在不同的行上。块中最后一个语句不需要最后的;。
- Perl中变量名的大小写对perl没有意义。
- 印记并不指示变量作用域，甚至不指示变量类型。Perl中的\$只是一个项，类似于\$scalar、\$array[0]或\$hash{\$key}。
- Perl用lt、le、eq、ne、ge和gt比较字符串。
- 没有魔法块，不过参见PerlX::MethodCallWithBlock。
- Perl的子例程定义发生在编译阶段，所以如果有以下代码：

```
use v5.10;
sub foo { say "Camelia" }
foo();
sub foo { say "Amelia" };
foo();
```



这会打印两次Amelia，因为运行阶段语句执行之前的最后一个定义才有效。这还说明在文件中子例程调用可以比子例程定义更早出现。

- Perl没有类变量，不过人们会用词法作用域变量来模拟类变量。
- Perl中的range操作符返回一个列表，不过参见PerlX::Range。
- /s模式修饰符使Perl的.可以匹配换行符，而Ruby使用/m达到同样的目的。Perl中/m使^和\$锚分别匹配逻辑行的开始和结束。
- Perl会扁平化列表。
- Perl中只要能放逗号的位置都可以放=>，所以你经常会看到Perl程序员使用箭头来指示方向：

```
rename $old => $new;
```

- 在Perl中，0、"0"、""、()和undef在布尔上下文中都是false。基本Perl并不需要一个特殊的布尔值。你可能希望使用boolean模块。
- Perl通常用undef来模拟nil的工作。
- Perl允许你稍有些马虎，因为有些字符不那么特殊。例如，变量后面的?不会对变量做任何处理：

```
my $flag = $foo? 0 :1;
```

## Java陷阱

- Perl中没有main，或者更确切地讲，没有：

```
public static void main(String[ ] argv) throws IOException
```
- Perl会在你需要时按需扩展数组和散列来分配内存。自动生成（autovivification）意味着如果为变量赋值，就会为它留出空间。
- Perl不允许提前声明变量，除非有use strict。
- 在Perl中，一个东西和它的引用是有差别的，所以（通常）必须显式地对后者解引用。
- Perl中并不是所有函数都必须是方法。
- Perl中字符串和数值直接量通常不是对象，不过它们也可以是对象。
- Java程序员也许希望用一个类来定义一个数据结构，可能会奇怪地发现Perl定义数据结构的方法完全不同，它会利用简单的数据声明，并混合匿名散列和数组来构建所有数据结构。参见第9章。
- Perl对象的实例数据（通常）只是用于这个对象的散列中的一个值，这个散列字段的名字对应于Java中实例数据的名字。

- Perl中私有性是可选的。
- 方法最后调用的函数直到运行时才能确定，任何对象或类只要有同名的方法，在Perl看来都是可以的。重要的是接口。
- Perl支持操作符重载。
- Perl不支持按签名函数重载。参见CPAN上Class::Multimethod模块。
- Perl允许多重继承，不过这更对应于Java中的多个接口，因为Perl类只继承方法，而不继承数据。
- Java char不是一个抽象Unicode码点。它是一个UTF-16代码单元，这说明它要占两个Java char，还需要特殊编码，从而能够在Java的基本多文种平面（Basic Multilingual Plane, BMP）之外使用。相比之下，Perl字符就是一个抽象码点，并有意将它的内层实现对程序员隐藏。Perl代码会自动适用于Unicode的整个范围，甚至还可能超出这个范围。
- 与Java中不同，Perl的字符串直接量中可以包含直接量换行符。不过，通常最好还是使用“here”文档。
- 函数通常返回undef而不是通过产生异常来指示错误。如果你希望产生异常来指示错误，可以使用autodie pragma。
- Perl不使用命名参数。程序的参数会出现在每个函数的@\_ array中。不过，通常会为它们立即指定名字。如果想用更形式化的方式声明命名参数，可以查看CPAN中的Methods::Signatures模块。
- Perl所指的函数原型与Java中的函数原型工作完全不同。
- Perl支持按命名参数传递，这允许忽略可选的参数，而且参数顺序可以自由地重新安排。
- Perl的垃圾回收系统要基于引用计数，所以可以编写一个析构函数自动清理资源，如打开的文件描述符、数据库连接、文件锁等。
- Perl正则表达式不需要额外的反斜线。
- Perl有正则表达式直接量，编译器会在编译时对它进行编译和语法检查，并存储以提高效率。
- Perl中的模式匹配不会像Java的match方法那样悄悄地在模式上放置锚。Perl的匹配更像是Java的find方法。
- Perl模式可以有多个同名的捕获组。
- Perl模式可以递归。
- 完成不区分大小写的匹配时，Java模式需要一个特殊的选项来指定使用Unicode大小

写转换（casefolding），不过Perl模式默认地就使用Unicode大小写转换。另外，如果进行不区分大小写的匹配，Perl会使用完全大小写转换，而Java只使用简单大小写转换。

- 在Java模式中，经典字符类（如\w和\s）默认为只包括ASCII，它需要一个特殊的选项来升级为包括Unicode。Perl模式默认就包括Unicode，所以它需要一个特殊的选项将经典字符类（或POSIX类）降级为只处理传统ASCII。
- Java的JNI对应于Perl的XS，至少在精神上是对应的。Perl模块通常有对应的已编译C/C++模块，而Java很少有。
- 并不是所有代码都要用Perl重写。Perl提供了一些很容易的方法，可以使用反引号、system和管道式open调用你的系统的本机程序。

## 效率

尽管编程的目的大多可能只是让程序能够正确地工作，但有时你可能还希望Perl程序能做得更好。Perl提供了丰富的操作符、数据类型和控制构造，不过这些构造在速度和空间优化方面不一定很直观。Perl的设计中做了很多折中，这些决定埋藏在代码内部。一般地，你的代码越简单越短小，它运行的速度就越快，不过也有例外。这一节就是要帮助你让代码表现得更好。

如果你希望代码大大加快运行速度，可以利用第16章介绍的Perl编译器后端，或者把你的内循环重写为一个C扩展（这本书中没有介绍）。不过，在做具体工作之前，你应当先对程序完成性能测试（参见第17章），了解是否可以先做某些简单的调整。

需要说明，时间方面的优化有时可能需要以空间或程序员的效率为代价（从下面的冲突提示可以看到）。“它们水火不相容。”如果编程是件很容易的事情，又何必需要这么复杂的人类来写那么简单的东西呢？不是吗？

## 时间效率

- 使用散列而不是线性搜索。例如，不要查找@keywords来查看\$\_是不是一个关键字，应当如下构造一个散列：

```
my %keywords;
for (@keywords) {
    $keywords{$_}++;
}
```

这样只需要检查\$keyword{\$\_}是否为非0值就可以很快得出\$\_是否包含一个关键字。

- foreach或列表操作符中避免使用下标。使用下标不仅是一个额外的操作，而且如果你的下标变量恰好是浮点数（可能因为你完成了算术运算），还会发生从浮点数



到整数的一个额外的转换。对此还有更好的方法。可以考虑使用`foreach`、`shift`和`splice`操作。另外考虑声明`use integer`。

- 避免使用`goto`。它会从你的当前位置向外扫描来查找指定的标签。
- 如果能用`print`，就要避免使用`printf`。
- 避免使用`$&`和它的两个伙伴`$``和`$'`。它们在程序中的任何出现会导致所有匹配都要保存所搜索的字符串以备将来引用。不过，一旦已经出现，再多几个也无妨。Perl v5.10通过`/p`引入了逐个匹配的变量（参见第5章），所以你不用遭受痛苦，也不用完全放弃这些特性。
- 避免对字符串使用`eval`。对字符串（尽管不是一个`BLOCK`）的`eval`会强制每次都要重新编译。作为一个解析器，Perl解析器的速度很快，但这并不能说明多少问题。如今总会有一种更好的方法来实现你想做的事情。具体来讲，如果使用`eval`只是为了构造变量名，这种代码已经过时，因为现在完全可以使用符号引用直接做到同样的事情：

```
no strict "refs";
$name = "variable";
$$name = 7;           # 设置$variable为7
```

倒不是我们推荐这种做法，不过如果你找不到其他的方法，使用这个方法总比使用字符串`eval`要好一些。

- 短路选择通常比相应的正则表达式更快。所以：

```
print if /one-hump/ || /two/;
```

可能比下面的正则表达式快：

```
print if /one-hump|two/;
```

至少对于`one-hump`和`two`的某些值来说是这样。这是因为，优化器喜欢把一些简单的匹配操作提升到语法树中更高的部分，并用Boyer-Moore算法完成非常快速的匹配。而复杂的模式就无法利用到这个特性。

- 用`next if`尽早排除常见情况。像简单正则表达式一样，优化器也会利用这一点。而且避免不必要的工作肯定是有意义的。通常可以在完成`split`或`chop`之前丢掉注释行和空行：

```
while (<>) {
    next if /^#/;
    next if /^$/;
    chop;
    @piggies = split(/,/);
    ...
}
```

- 避免有多个量词的正则表达式，或者带小括号的表达式上有很大的`{MIN,MAX}`数。这

种模式可能导致速度极慢的回溯行为，除非定量子模式第一“趟”就匹配成功。还可以使用(`>...`)构造来强制一个子模式要么完全匹配，要么失败而不回溯。

- 尽可能增加正则表达式中非可选直接量字符串的长度。这一点有些违反我们的直觉，不过较长的模式通常确实比较短的模式匹配得更快。这是因为，优化器会查找常量字符串，把它们交给Boyer-Moore算法来完成搜索，字符串越长越受益。可以用Perl的`-Dr`调试开关编译你的模式，来看看在Perl眼里最长的直接量字符串是怎样的。
- 避免紧凑循环中有开销很大的子例程调用。调用子例程存在相关的开销，特别是传递很长的参数表或返回很长的值时，可以考虑采用以下方法：按引用传递值；将值作为动态作用域全局变量传递；内联子例程；或者用C重写整个循环，这些方法中比较倾向于前者，后面的方法一个比一个让人绝望（还有比所有这些解决方案更好的做法：可以使用一个更聪明的算法，从而使子例程不复存在）。
- 如果你知道每次都会得到相同的答案，就不要反复地调用同一个子例程。`Memoize` 等模块会有帮助，或者可以构造你自己的缓存，一旦有了答案（而且你知道它不会改变），就建立缓存。
- 对于`getc`，除了完成单字符终端I/O外，避免它做任何其他事情。实际上，其他情况下根本不要用`getc`，而应当使用`sysread`。
- 避免对长字符串频繁地调用`substr`，特别是这个字符串中包含UTF-8时。在字符串开头使用`substr`是可以的。对于某些任务，可以利用一个4参数的`substr`在处理过程中“吞掉”字符串，并用`"`替换你捕获的部分，从而保持在字符串的开头调用`substr`。

```
while ($buffer) {  
    process(substr($buffer, 0, 10, ""));  
}
```

- 如果可以，要用`pack`和`unpack` 而不是使用多个`substr`调用。
- 使用`substr`作为左值，而不是联接子串。例如，要把`$foo`的第4到第7个字符替换为变量`$bar`的内容，不要这么做：

```
$foo = substr($foo,0,3) . $bar . substr($foo,7);
```

实际上，只需要确定要替换的字符串部分，再为它赋值就可以了，如下所示：

```
substr($foo, 3, 4) = $bar;
```

不过要当心，如果`$foo`是一个庞大的字符串，而`$bar`没有达到这个“洞”的长度，这可能导致大量复制。Perl会从最前面或最后面复制来尽量减少这种复制，不过如果`substr`在中间，它就没有其他办法了。

- 使用`s///`而不是联接子串。如果可以把一个常量替换为另一个相同大小的常量，这一点尤其适用。这会产生一个原地替换。
- 使用语句修饰符以及等价的`and`和`or`操作符，而不是完整的条件。语句修饰符（如

`$ring = 0 unless $engaged`) 和逻辑操作符可以避免进入和离开一个块的开销，而且它们通常更可读。

- 使用`$foo = $a || $b || $c`。这比下面的代码更快（也更简短）：

```
if ($a) {  
    $foo = $a;  
}  
elsif ($b) {  
    $foo = $b;  
}  
elsif ($c) {  
    $foo = $c;  
}
```

类似地，可以如下设置默认值：

```
$pi ||= 3;
```

- 将希望得到相同初始字符串的所有测试归组在一起。测试一个字符串查看其中各个前缀时（类似于switch结构），可以把所有`/^a/`模式放在一起，所有`/^b/`模式放在一起，如此等等。
- 不要测试你很清楚根本不会匹配的字符串。要使用`last`或`elsif`，避免落到switch语句的下一个分支。
- 使用特殊操作符（如`study`）、逻辑字符串操作符以及`pack "u"`和`unpack "%"`格式。
- 注意“防微杜渐”。类似`(<STDIN>)[0]`的错误语句可能导致Perl做很多不必要的工作。同样按照Unix的哲学，Perl允许你积少成多，直到最后完全崩溃。
- 把操作拿到循环外面。Perl优化器并不会从循环中删除不变的代码。它希望你自己能有所认识。
- 字符串可能比数组快。
- 数组可能比字符串快。这取决于你是否打算重用字符串或数组，以及你要完成哪些操作。如果要大量修改各个元素，这意味着数组会好些，如果只是偶尔修改某些元素，则意味着字符串更好些。不过你要自己试试看。
- `my`变量比`local`变量更快。
- 对生成的键数组排序会比使用一个复杂的`sort`子例程更快。给定的数组值通常会比较多次，所以如果`sort`子例程必须大量重新计算，最好还是在排序之前把这个计算分出来单独执行。
- 如果要删除字符，`tr/abc//d`比`s/[abc]//g`快。

带逗号分隔符的`print`可能比联接字符串快。例如：



```
print $fullname{$name} . " has a new home directory " .
    $home{$name} . "\n";
```

这里要把两个散列和两个固定字符串联接在一起，然后再把它们传递到底层print例程，而：

```
print $fullname{$name}, " has a new home directory ",
    $home{$name}, "\n";
```

则不用。另一方面，取决于不同的值和体系结构，联接字符串也可能更快。可以试试看。

- `join("", ...)`要优于一系列联接的字符串。多个联接可能会导致来回多次复制字符串。`join`操作符可以避免这一点。
- 对固定字符串的`split`通常比模式的`split`更快。也就是说，如果你知道只有一个空格，应当使用`split(/ /, ...)`而不是`split(/ +/, ...)`。不过，模式`/\s+/, /\^/, //`和`/ /`得到了特殊优化，类似于对空白符的特殊`split`。
- 预扩展一个数组或字符串可能会节省一些时间。随着字符串和数组的扩展，Perl会分配一个新副本，其中留出扩展的空间，并把原来的值复制进去，以此完成扩展。可以利用`x`操作符预扩展一个字符串，或者通过设置`$#array`预扩展一个数组，从而避免这种非经常性的开销，并减少内存碎片。
- 如果长字符串和数组会重用于同样的用途，不要把它们置为`undef`。这有助于避免重新扩展字符串或数组时的重新分配。
- `"\0" x 8192`优于`unpack("x8192",())`。
- 如果`mkdir`系统调用不可用，`system("mkdir ...")`建立多个目录可能更快。
- 如果返回值已经指示到达末尾，则避免使用`eof`。
- 缓存文件中（如`passwd`和`group`文件）很可能重用的项。缓存从网络得到的项尤其重要。例如，将数字地址（如204.148.40.9）转换为地址名（“www.oreilly.com”）时，要缓存来自`gethostbyaddr`的返回值，可以使用以下代码：

```
sub numtoname {
    local ($) = @_;
    unless (defined $numtoname{$_}) {
        my(@a) = gethostbyaddr(pack("C4", split(/\./)),2);
        $numtoname{$_} = @a > 0 ? $a[0] : $_;
    }
    return $numtoname{$_};
}
```

- 避免不必要的系统调用。操作系统调用往往开销都比较大。所以，例如如果可以利用`$now`的一个缓存值，就不要调用`time`操作符。可以使用特殊的\_文件句柄来避免不必要的`stat(2)`调用。在某些系统上，甚至一个最小的系统调用都会执行上千个指令。

- 避免不必要的system调用。system函数必须派生一个子进程来执行你指定的程序，或者更糟糕的是，可能要运行一个shell来执行你的程序。这很容易执行上百万个指令。
- 当心子进程的启动，不过只有当它们会频繁调用时才应注意。启动一个*pwd*、*hostname*或*find*进程并没有太大危害，毕竟shell一直在启动子进程。我们有时确实鼓励使用这种工具箱方法，信不信由你。
- 自行跟踪你的工作目录，而不是反复调用*pwd*（为此提供了一个标准模块，参见第28章中的Cwd）。
- 避免命令中使用shell元字符，适当情况下应当向system和exec传递列表。
- 在不要求分页的机器上为Perl解释器设置粘贴位：  

```
% chmod +t /usr/bin/perl
```
- 将系统调用替换为非阻塞的管道式open。输入到来时读取输入，而不是等待整个程序完成后再读取。
- 使用异步事件处理（AnyEvent、Coro、POE、Gearman等）来同时完成多个工作。现代机器可能有多个CPU，而且这些CPU有可能是多核的。有些事件只是在等待网络响应，这会阻塞你的程序做其他事情。有些异步事件处理可以代替阻塞的system调用。

## 空间效率

- 为变量提供最短的作用域，以免不需要它们时还要占用空间。
- 如果整数定宽，使用vec提供紧凑的整数数组存储（宽度可变的整数可以用UTF-8字符串存储）。
- 尽量用数值而不是等价的字符串值；数值占用的内存更少。
- 使用substr存储较长字符串中的定长字符串。
- 如果键和值的长度是固定的，使用Tie::SubstrHash模块提供非常紧凑的散列数组存储。
- 使用\_\_END\_\_和DATA文件句柄避免将程序数据同时存储为字符串和数组。
- 如果顺序不重要，尽量使用each而不是keys。
- 如果全局变量不再使用，将其删除或置为undef。
- 使用某种DBM在磁盘上存储散列，而不是存储在程序中。
- 使用临时文件存储数组。
- 使用管道将处理工作分摊到其他工具。它们退出时会清理其内存使用。

- 避免列表操作和读取整个文件。
- 避免使用`tr///`。每个`tr///`表达式必须存储一个很庞大的转换表。
- 不要展开循环或内联子例程。
- 如果不需要修改数据（有时甚至在需要修改数据时），使用`File::Mmap`读取文件。
- 避免递归。Perl没有尾调用优化，因为它是一种动态语言。你应当能够将这些递归转换为一个迭代方法，提供尾调用优化递归的语言采用的也这种迭代方法。

## 程序员效率

如果你能在今天运行一个不算完美的程序，这也比下个月才能运行一个最完美的程序要好。处理一些临时“丑陋性”<sup>注1</sup>。以下有些建议与我们前面给出的建议正好相反。

- 写你自己的代码前先查看CPAN。
- 反复查看CPAN。你可能会漏掉你需要的模块。请仔细查看。
- 使用默认值。
- 使用特别的快捷命令行开关，如`-a`、`-n`、`-p`、`-s`和`-i`。
- 使用`for`表示`foreach`。
- 用反引号运行系统命令。
- 使用`<*>` 等。
- 使用运行时创建的模式。
- 在模式中自由使用`*`、`+`和`{}`。
- 处理整个数组和读取整个文件。
- 使用`getc`。
- 使用`$``、`$&`和`$'`。
- 不要检查`open`的错误值，因为给定一个非法句柄时，`<HANDLE>`和`print HANDLE`会表现为一个无操作（no-op）。
- 不要关闭文件，下一个`open`时文件会自行关闭。
- 不要传递子例程参数。要使用全局变量。
- 不要对子例程参数命名。可以用`$_[EXPR]`直接访问这些参数。
- 使用你最先想到的方法。

---

注1：这也称为“技术债务”，不过这并不总是坏事。



- 让别人为你工作，可以只写一半实现，再把它放在Github上（让别人继续完成）。

## 维护人员效率

如果要编写你（或你的朋友）将要使用和维护很长时间的代码，那么写代码时一定要更多地考虑到将来。不要只着眼于一些短期的收益，而应立足于更大的长期收益。

- 不要使用默认设置。
- 要表示foreach就用foreach，而不要使用for。
- 结合next和last使用有意义的循环标签。
- 使用有意义的变量名。
- 使用有意义的子例程名。
- 使用and、or和语句修饰符（如exit if \$done）把重要的东西放在行首。
- 一旦处理完文件就将其关闭。
- 使用包、模块和类来隐藏你的实现细节。
- 建立合理的API。
- 传递实参作为子例程参数。
- 使用my对子例程参数命名。
- 加小括号使代码更清晰。
- 加入大量（有用的）注释。
- 包含嵌入的pod文档。
- 使用use warnings。
- 使用use strict。
- 写测试并保证合适的测试覆盖度（参见第19章）。

## 移植人员效率

- 给足小费。
- 避免使用还没有实现的函数。可以用eval测试来查看有哪些可用函数。
- 使用Config模块或\$^O变量来确定你运行的机器是哪种类型。
- 加入use v5.xx语句来指示你需要的Perl。
- 不要只是为了追求时髦而使用新特性。

- 不要希望本机浮点数和双精度数会在其他机器上完成pack和unpack。
- 在网络上发送二进制数据时使用网络字节序（pack的“n”和“N”格式）。
- 不要在网络上发送二进制数据。应当发送ASCII。更好的做法是发送UTF-8。不过，最好还是直接送钱。
- 在语言或服务未知的情况下，使用标准或常用格式（如JSON或YAML）来完成数据交换。
- 检查\$]或\$^V，查看当前版本是否支持你使用的所有特性。
- 不要使用\$]或\$^V。应当结合版本号使用require或use。
- 即使不使用，也要加入eval exec，这样你的程序就能在那些有类Unix shell但不能识别#!记法的系统上运行（尽管这种系统很少）。
- 即使不使用，也要加入#!/usr/bin/perl行。
- 测试Unix命令的变体。例如，有些find程序不能处理-xdev开关。
- 如果可以在内部完成，则避免Unix命令的变体。Unix命令在MS-DOS或VMS上不能很好地工作。
- 把你的所有脚本和手册页放在所有机器都装载的一个网络文件系统中。
- 将你的模块发布到CPAN上。如果你的程序不可移植，你会得到很多反馈。
- 通过使用公共源代码控制（如Github），使别人可以很容易地为你工作做出贡献。

## 用户效率

与让你自己轻松相比，要让别人轻松，你还要做更多的工作。

- 不要让用户逐行地输入数据，可以为用户弹出他们喜欢的编辑器来完成输入。
- 更好的做法是，可以使用一个GUI（如Perl /Tk或Wx模块），用户可以在GUI中控制事件的顺序。
- 在你继续工作的同时，提供一些东西让用户阅读。
- 使用自动加载，使程序看起来运行得更快。
- 每个提示符上可以提供一些有帮助的消息。
- 如果用户没有提供正确的输入，可以提供一个有帮助的用法消息。
- 在文档中包含扩展的例子，并在发行版本中提供完整的示例程序。
- 每个提示符上显示默认动作，可能还可以提供一些候选选择。
- 为初学者提供默认值，并允许专家修改这些默认值。

- 在适当的情况下使用单字符输入。
- 按用户熟悉的方式完成交互。
- 使错误消息清楚地指出哪里需要修正。包括所有相关信息，如文件名和错误代码，如下：

```
open(FILE, $file) || die "$0: Can't open $file for reading: $!\n";
```

- 如果其余脚本只是批处理，使用`fork && exit`脱离终端。
- 允许参数来自命令行或标准输入。
- 使用的配置文件要采用一种简单的文本格式。CPAN上已经有很多这样的模块。
- 不要在程序中随意施加限制。
- 尽量使用变长字段而不是定长字段。
- 使用面向文本的网络协议。
- 告诉所有人使用面向文本的网络协议！
- 告诉所有人进一步告诉所有其他人使用面向文本的网络协议！
- 懒要懒得有共鸣。
- 要好心为用户考虑。

## 有风格的编程

在格式化方面，你肯定有你自己的喜好，不过有一些通用的原则可以让你的程序更容易阅读、理解和维护。Larry在`perlstyle`中给出了一些通用建议，不过它们只是建议。你可能还想看看《Perl Best Practices》或《Modern Perl》。

最重要的一点是：要在`strict`和`warnings pragma`下运行你的程序，除非有充分的理由不这么做。如果需要将这些`pragma`关闭，要在尽可能小的范围内使用`no`。`sigtrap`甚至`diagnostics pragma`也会有帮助。

关于代码布局的美感，Larry唯一关心的一点就是多行`BLOCK`的结束大括号应当“外凸”，与这个构造的起始关键字行对齐。除此以外，他还有其他一些偏好，不过没有这么强调。这本书中的例子（应当）都遵循以下编码约定：

- 使用4列缩进。
- 开始大括号与其前面的关键字尽可能放在同一行上；否则，将它们垂直对齐：

```
while ($condition) { # 如果这一行比较短，则与关键字放在同一行上
    # 做些处理
}
```



```
# 如果条件有多行，则大括号相互对齐
while ($this_condition and $that_condition
      and $this_other_long_condition)
{
    # 做些处理
}
```

- 在多行BLOCK的开始大括号前加空格。
- 短BLOCK可以写在一行上，包括大括号。
- 简短的单行BLOCK省略分号。
- 大多数操作符两边都要加空格。
- 在“复杂”的下标（中括号内）两边加空格。
- 在完成不同工作的代码块之间加空行。
- 在结束大括号和else之间加换行符。
- 不要在函数名和它的开始小括号之间加空格。
- 不要在分号前加空格。
- 每个逗号后面要加空格。
- 对于长的代码行，要在操作符后断行（但是要在and和or之前，有时这也写作&&和||）。
- 将相应的项垂直对齐。
- 只要能清楚地表达，省略冗余的标点符号。

Larry对于以上各个约定都有自己的理由，不过他并没有要求所有人都像他一样考虑问题。

以下还有一些需要考虑的更实在的风格问题：

- 只是因为你可以用某种特定方式做某件事，并不意味着你就应该那么做。按照Perl的设计，会为你提供很多方法来完成某个事情，所以你要考虑选择其中可读性最好的一个。例如：

```
open(F00,$foo) || die "Can't open $foo: $!";
```

就比下面的方法要好：

```
die "Can't open $foo: $!" unless open(F00,$foo);
```

因为第二个方法把语句的要点隐藏在一个修饰符中。

另一方面：

```
print "Starting analysis\n" if $verbose;
```

要优于：

```
$verbose and print "Starting analysis\n";
```

因为关键不是用户是否键入-v。

- 类似地，只是因为一个操作符允许你假设有默认参数，并不意味着你必须使用这些默认参数。默认参数是为只写一次性程序的懒程序员提供的。如果你希望你的程序可读，就应该考虑提供参数。
- 在同一行上可以在很多地方省略小括号，但这并不意味着你应该这么做：

```
return print reverse sort num values %array;  
return print(reverse(sort num (values(%array))));
```

如果有疑问，就应该加小括号。至少，这会让一些可怜的家伙在vi中注意到%键。

即使你没有疑问，也要考虑到在你之后维护这个代码的人，另外还要考虑到可能有人把小括号放错了位置。

- 不要错误地以为只能在顶端或底部退出循环。Perl提供了last操作符，可以在中间退出。你还可以让它“外凸”，使它更明显：

```
LINE:  
    for (;;) {  
        statements;  
        last LINE if $foo;  
        next LINE if /^#/;  
        statements;  
    }
```

- 不要害怕使用循环标签。它们的用途是为了提高可读性，同时还可以允许终止多层循环。参见前面给出的例子。
- 避免在void上下文中（也就是说，需要丢掉返回值时）使用grep、map或反引号。这些函数都有返回值，所以应当使用它们的返回值。否则，应该使用foreach循环或system函数。
- 为保证可移植性，使用并非所有机器都实现的特性时，要在一个eval中测试这个构造，查看它是否失败。如果你知道一个特定特性的版本或补丁级别，可以测试\$]（English模块中的\$PERL\_VERSION），来看是否有这个特性。Config模块也允许查询安装Perl时Configure程序确定的值。
- 选择有助于记忆的标识符。如果你不能记住它的含义，就会有问题。
- 尽管像\$gotit这样的短标识符可能没问题，不过应当使用下划线来分隔单词。\$var\_names\_like\_this通常比\$VarNamesLikeThis更易读，特别是对那些英语不是母语的人来说。另外，这个原则也同样适用于\$VAR\_NAMES\_LIKE\_THIS。

对于这个规则，有时包名是个例外。Perl非正式地保留了小写的模块名，用于那些作

为pragma的模块，如integer和strict。其他模块都应当以一个大写字母开头，并使用大小写混合，不过可能由于某些原始文件系统的名字长度限制而省略下划线。

- 你会发现使用字母大小写来指示一个变量的作用域或性质很有好处。例如：

```
$ALL_CAPS_HERE    # 仅常量（注意不要与Perl变量冲突）
$Some_Caps_Here   # 包作用域内全局/静态变量
$no_caps_here     # 函数作用域内my()或local()变量
```

出于一些不明确的原因，函数和方法最好都用全小写。例如，`$obj->as_string()`。

可以使用一个前导下划线来指示一个变量或函数不能在定义它的包外使用（Perl没有强制这一点，这只是一种文档形式）。

- 如果有一个非常复杂的正则表达式，可以使用/x修饰符，并加入一些空白符，使它看起来更有意义，不像线路噪声那么晦涩。
- 如果你的正则表达式已经有太多斜线或反斜线，不要使用斜线作为分隔符。
- 如果你的字符串包含引号，不要使用相同类型的引号作为定界符。应当使用q//、qq//或qx//伪函数。
- 使用and和or操作符避免对列表操作符加太多小括号，并减少&&和||等标点符号操作符的影响范围。要把子例程当作函数或列表一样来调用，以避免过多的&和小括号。
- 使用here文档而不要反复使用print语句。
- 将相应的项垂直对齐，特别是太长不能放在一行上时：

```
$IDX = $ST_MTIME;
$IDX = $ST_ETIME if $opt_u;
$IDX = $ST_ETIME if $opt_c;
$IDX = $ST_SIZE  if $opt_s;
mkdir($tmpdir, 0700) || die "can't mkdir $tmpdir: $!";
chdir($tmpdir)      || die "can't chdir $tmpdir: $!";
mkdir("tmp", 0777)  || die "can't mkdir $tmpdir/tmp: $!";
```

- 重复三次就是真理：

一定要检查系统调用的返回码。

一定要检查系统调用的返回码。

一定要检查系统调用的返回码！

错误消息应当输出到STDERR，而且应当指出哪个程序导致了这个问题，以及失败的调用及其参数是什么。最重要的是，对于失败的系统调用，消息应当包含标准系统错误消息，指出哪里出了问题。下面是一个简单但很完整的例子：

```
opendir(D, $dir) || die "Can't opendir $dir: $!";
```

记住，一定要检查返回码。



- 在适当的情况下将转换（transliterations）对齐：

```
tr [abc]
    [xyz];
```

- 考虑可重用性。如果你希望再做类似的事情，为什么要浪费精力去写一次性的脚本呢？可以考虑将你的代码通用化。可以考虑编写一个模块或对象类。可以考虑让你的代码在`use strict`和`-w`起作用时简洁地运行。可以考虑发布你的代码。可以考虑改变你的整个世界观。可以考虑……呵，开玩笑的。
- 使用`Perl::Tidy`美化代码，使用`Perl::Critic`捕获可能的编程问题。
- 要一致。
- 要体贴。

## 老练的Perl

前面几节（以及前面的各章）中我们已经接触了一些惯用法，不过查看Perl程序员完成的程序时，你还会经常看到很多其他惯用法。在这里谈到Perl的惯用法时，并不是指一组固化的Perl表达式。实际上，我们是指一些Perl代码，这些Perl代码可以展示对语言流的理解，要避开什么，何时避开，另外可以得到什么以及何时得到。

我们不可能列出你看到的所有惯用法，这需要整本书才能做到，甚至可能两本书才行（例如，参见《Perl Cookbook》）。不过下面给出一些重要的惯用法，这里的“重要”定义为“对于那些自认为对计算机语言应该如何工作已经很了解的人，可能会让他们惊诧的做法”。

- 在你认为有助于提高可读性的地方使用`=>`取代逗号：

```
return bless $mess => $class;
```

这会读作，“祝福这个`mess`为指定的类”。不过要注意，不要在你不想自动加引号的词后面使用`=>`：

```
sub foo () { "FOO" }
sub bar () { "BAR" }
print foo => bar;    # 打印fooBAR,而不是FOOBAR;
```

还有一个地方很适合使用`=>`：可以在一个看起来容易混淆的直接量逗号附近使用`=>`：

```
join(", " => @array);
```

Perl会为解决问题提供多种方法，所以你可以充分锻炼你的能力，力求创新。多多练习吧！

- 使用单数代词来提高可读性：

```
for (@lines) {
    $_ .= "\n";
}
```

`$_` 变量是Perl版本的代词，实际上表示“它”。所以上面的代码就表示“对于每一行，为它追加一个换行符”。如今甚至可以写为：

```
$_ .= "\n" for @lines;
```

`$_` 代词对于Perl非常重要，所以在`grep`和`map`中会强制使用。下面给出一种建立缓存的方法，可以缓存开销很大的函数的常见结果：

```
%cache = map { $_ => expensive($_) } @common_args;
$val = $cache{$x} || expensive($x);
```

- 省略代词进一步提高可读性<sup>注2</sup>。
- 结合语句修饰符使用循环控制。

```
while (<>) {
    next if /^=for\s+(index|later)/;
    $chars += length;
    $words += split;
    $lines += y/\n//;
}
```

我们使用这个代码片段统计这本书的页数。如果要对同一个变量做大量处理，通常更可读的做法是完全省略代词，这与一般看法正好相反。

这个代码片段还展示了结合语句修饰符使用`next`来短路循环的惯用法。

`grep`和`map`中的循环控制变量总是`$_`变量，不过程序对它的引用通常是隐式的：

```
@haslen = grep { length } @random;
```

这会取一个随机标量列表，只选出其中长度大于0的标量。

- 使用`for`为代词建立先行词：

```
for ($episode) {
    s/fred/barney/g;
    s/wilma/betty/g;
    s/pebbles/bambam/g;
}
```

那么，如果循环中只有一个元素呢？这是一个设置“它”（即`$_`）的便利方法。从语言学角度讲，这称为主题化。这不是窃取，而是一种交流。

- 隐式地引用复数代词`@_`。
- 使用控制流操作符来设置默认值：

注2： 这一节中，连续几个要点都使用后面的同一个例子，因为有些例子可以展示多个惯用法。

```

sub bark {
    my Dog $spot = shift;
    my $quality = shift || "yapping";
    my $quantity = shift || "nonstop";
    ...
}

```

这里我们隐式地使用了另一个Perl代词@\_，这表示“它们”。函数的参数总是以“它们”形式出现。即使省略这个代词，shift操作符也知道要处理@\_，就像迪斯尼乐园的过山车操作员可能总会叫“下一组！”，即使没有指定哪个队前移（指定哪一队也是没有意义的，因为这里只有一队在移动）。

可以用||设置默认值，尽管它原本是一个布尔操作符，因为Perl会返回第一个true值。Perl程序员通常对真值表现出一种漫不经心的态度，例如，如果你想指定数量为0，上面这一行会有问题。不过，只要你不想把\$quality或\$quantity设置为一个false值，这个惯用法就很不错。如果先纳入所有可能性，然后再到处放上defined和exists调用，这是没有意义的。你必须了解在做什么。只要不会偶尔为false，那就没有问题。

如果你认为它偶尔会为false，可以使用已定义的OR操作符//：

```

use v5.10;

sub bark {
    my Dog $spot = shift;
    my $quality = shift // "yapping";
    my $quantity = shift // "nonstop";
    ...
}

```

- 使用赋值形式的操作符，包括控制流操作符：

```
$xval = $cache{$x} ||= expensive($x);
```

这里根本没有初始化缓存。我们只是依赖 ||= 操作符来调用expensive(\$x)，并仅当\$cache{\$x}为false时将它赋至\$cache{\$x}。其结果是\$cache{\$x}的新值。同样的，对于真值我们采用了“随意”的方法，如果缓存了一个false值，则会再次调用expensive(\$x)。程序员可能知道这是可以的，因为expensive(\$x)返回false时开销并不大，或者也许程序员知道expensive(\$x)根本不会返回一个false值。或者有可能程序员只是很随意。随意也可以算是一种创造性。

- 使用循环控制作为操作符，而不只是语句。而且……
- 像小分号一样使用逗号：

```

while (<>) {
    $comments++, next if /^#/;
    $blank++, next if /^\s*$/;
    last if /^_ _END_ _/;
    $code++;
}

```



```

}
print "comment = $comments\nblank = $blank\ncode = $code\n";

```

这里展示了这样一种认识：语句修饰符会修改语句，而`next`只是一个操作符。它还显示了逗号常用于分隔表达式，就像正常使用一个分号一样（区别在于，逗号使两个表达式作为同一个语句的一部分，在同一个语句修饰符控制之下）。

- 充分利用流控制：

```

while (<>) {
    /^#/ and $comments++, next;
    /^s*$/ and $blank++, next;
    /^_ _END_ _/ and last;
    $code++;
}
print "comment = $comments\nblank = $blank\ncode = $code\n";

```

这里还是同一个循环，只是这一次模式在前面。老练的Perl程序员知道这与前一个例子一样，也会编译得到同样的内部代码。`if`修饰符就是一个反向`and`（或`&&`）连接，而`unless`修饰符只是一个反向`or`（或`||`）连接。

- 使用`-n`和`-p`开关提供的隐式循环。
- 不要在单行块末尾加分号：

```

#!/usr/bin/perl -n
$comments++, next LINE if /^#/;
$blank++, next LINE if /^s*$/;
last LINE if /^_ _END_ _/;
$code++;

END { print "comment = $comments\nblank = $blank\ncode = $code\n" }

```

这与前面的程序基本相同。这里把一个显式的`LINE`标签放在循环控制操作符上，因为我们喜欢这样做，不过并不一定要这么做，因为`-n`提供的隐式`LINE`循环是最内层循环。

我们使用一个`END`使最后一个`print`语句在隐式主循环之外，这与`awk`中类似。

- 如果有太多`print`，则使用`here`文档。
- 在`here`文档中使用一个有意义的定界符：

```

END { print <<"COUNTS" }
comment = $comments
blank = $blank
code = $code
COUNTS

```

熟练的Perl程序员不会使用多个`print`，而会使用一个带内插的多行字符串。尽管我们之前把它称为常见错误，不过这里已经省略了末尾的分号，因为`END`块的末尾没有必要加分号（如果要把它变成一个多行的块，可以再加上分号）。

- 对标量顺带完成替换和转换：

```
($new = $old) =~ s/bad/good/g;
```

或者使用/r修饰符返回结果：

```
$new = $old =~ s/bad/good/gr;
```

由于作为左值可以修改，所以可以这么说：你会经常看到人们在赋一个值时“顺带地”改变这个值。这实际上可以在内部节省一个字符串复制（如果我们想要实现这个优化）：

```
chomp($answer = <STDIN>);
```

原地修改参数的函数也可以利用这种“顺带”技巧。

不过等一下，还有呢！

- 不要局限于只是顺带地修改标量：

```
for (@new = @old) { s/bad/good/g }
```

这里我们把@old复制到@new，顺带地改变所有内容（当然，不是一次完成，这个块会反复执行，一次改变一个“它”）。

- 使用特别的=>逗号操作符传递命名参数。
- 依赖于散列赋值来完成奇/偶参数处理：

```
sub bark {
    my DOG $spot = shift;
    my %parm = @_;
    my $quality = $parm{QUALITY} || "yapping";
    my $quantity = $parm{QUANTITY} || "nonstop";
    ...
}

$fido->bark( QUANTITY => "once",
            QUALITY => "woof" );
```

命名参数通常算是尚能承受的奢侈品。对于Perl，命名参数几乎没有开销（如果不考虑散列赋值的开销）。

- 重复布尔表达式直到为false。
- 适当情况下使用最小匹配。
- 使用/e修饰符计算替换表达式：

```
#!/usr/bin/perl -p
1 while s/^(.*?)(\t+)/$1 . " " x (length($2) * 4 - length($1) % 4)/e;
```

这个程序修正你从别人那里接收到的文件，这个人可能错误地认为他可以重新定义硬件tab来占4个空格而不是8个空格。这里利用了很多重要的惯用法。首先，如果你

在循环中要做的所有工作实际上都由条件语句完成，这种情况下，`1 while`用法就很方便（Perl很聪明，它知道在void上下文中不用警告你使用了`1`）。我们必须重复这个替换，因为每次用一定数目的空格替换tab时，都必须从头开始重新计算下一个tab的列位置。

`(.*?)`匹配第一个tab前它能找到的最小字符串，这里使用了最小匹配修饰符（问号）。我们也可以使用一个普通的贪婪`*`，比如`([^\t]*)`。不过，这样做之所以可行只是因为tab是单个字符，所以可以使用一个否定字符类来避免越过第一个tab。一般来讲，最小匹配修饰符更妥善，即使下一个要匹配的符号长度大于一个字符也不会有问题。

`/e`修饰符使用一个表达式而不是简单的字符串来完成替换。这允许我们根据需要立即完成计算。

- 对复杂的替换使用创造性的格式化和注释：

```
#!/usr/bin/perl -p
1 while s{
    ^                # 锚定在开始位置
    (                # 开始第一个子组
        .*?          # 匹配最小数目的字符
    )                # 结束第一个子组
    (                # 开始第二个子组
        \t+          # 匹配一个或多个tab
    )                # 结束第二个子组
}
{
    my $spacelen = length($2) * 4; # 计算完整tab的长度
    $spacelen -= length($1) % 4;   # 计算非偶数tab的长度
    $1 . " " x $spacelen;         # 生成正确数目的空格
}ex;
```

这可能有些“大材小用”了，不过有些人认为这比原先的单行代码更震撼。你可以试试看。

- 如果你喜欢，可以继续使用`$``：

```
1 while s/(\t+)/" " x (length($1) * 4 - length($`) % 4)/e;
```

这里是一个简短版本，它使用了`$``，我们已经知道这会影响性能。不过我们只是使用它的长度，所以情况并不那么糟糕。

- 直接使用`@-`（`@LAST_MATCH_START`）和`@+`（`@LAST_MATCH_END`）数组的偏移量：

```
1 while s/\t+/" " x (($+[0] - $-[0]) * 4 - $-[0] % 4)/e;
```

这个版本更简短（如果没有看到数组，可以试着查找数组元素）。参见第25章中的`@-`和`@+`。

- 结合常量返回值使用`eval`：



```

sub is_valid_pattern {
    my $pat = shift;
    return eval { "" =~ /$pat/; 1 } || 0;
}

```

不必使用`eval { }`操作符来返回一个真正的值。如果到达末尾我们总是返回1。不过，如果`$pat`中包含的模式有问题，`eval`会捕获问题，并向`||`操作符的布尔条件返回`undef`，这会将它转换为一个已定义的0（只是出于礼貌，因为`undef`也是`false`，不过`undef`可能会让某些人认为`is_valid_pattern`子例程出问题了，我们不希望这样，不是吗）。

- 使用模块完成所有具体工作。
- 使用对象工厂。
- 使用回调。
- 使用堆栈来跟踪上下文。
- 使用负下标访问数组或字符串末尾：

```

use XML::Parser;

$p = XML::Parser->new( Style => "subs" );
setHandlers $p Char => sub { $out[-1] .= $_[1] };

push @out, "";

sub literal {
    $out[-1] .= "C<";
    push @out, "";
}

sub literal_ {
    my $text = pop @out;
    $out[-1] .= $text . ">";
}

...

```

这是从一个250行程序中抽取的代码片段，我们使用这个程序将老骆驼书的XML版本转换回pod格式，所以可以先用一个真正的文本编辑器（Real Text Editor）编辑这个版本，然后再把它转换回DocBook。

首先你会注意到，这里依赖`XML::Parser`模块（来自CPAN）来正确解析XML，所以我们不用了解这是如何做到的。这样就直接减少了几千行代码（假设我们要用Perl重新实现`XML::Parser`为我们提供的一切<sup>注3</sup>，包括从几乎任何字符集到UTF-8的转换）。

`XML::Parser`使用了一个高级惯用法，称为对象工厂（object factory），在这里就是

---

注3： 实际上，`XML::Parser`只是一个复杂的包装器，它包装了James Clark的expat XML解析器。

一个解析器工厂。创建一个XML::Parser对象时，会告诉它我们想要哪种风格的解析器接口，它会为我们创建这样一个解析器。如果你不清楚长远看来哪种接口最好，这就是构建一个试验床应用的绝妙方法。subs风格只是XML::Parser的一种接口。实际上，这是最老的接口之一，可能甚至不算是当前最流行的一个接口。

setHandlers行显示了这个解析器的一个方法调用，没有采用箭头记法，而是采用了“间接对象”记法，允许省略参数两边的小括号（另外还有其他一些方面）。这一行还使用了我们之前介绍过的命名参数惯用法。

这一行还显示另外一个很强大的概念：回调。我们不是调用解析器来得到下一项，而是告诉它来调用我们。对于类似<literal>的命名XML标记，这个接口会自动调用一个同名的子例程（或者对于相应的结束标记，要在这个名字后面再加一个下划线）。不过，标记之间的数据没有名字，所以我们用setHandlers方法建立了一个Char回调。

接下来初始化@out数组，这是一个输出堆栈。我们在其中放入一个null字符串，来表示在当前标记嵌入层（开始为0）还没有收集任何文本。

现在要引入回调了。看到文本时，它会通过回调中的\$out[-1]惯用法自动追加到数组的最后一个元素。在外部标记级，\$out[-1]等同于\$out[0]，所以\$out[0]最后会给出整个输出（当然是最后才给出。不过首先我们必须处理标记）。

假设看到一个<literal>标记，将会调用literal子例程，将一些文本追加到当前输出，然后将一个新的上下文压入@out堆栈。现在直到结束标记的所有文本都已经追加到这个栈的新末尾。到达结束标记时，从@out栈弹出我们收集的\$text，并把变形后其余数据追加到栈的新末尾（也就是原来的末尾），结果就是把XML字符串<literal>text</literal>转换为相应的pod字符串C<text>。

其他标记的子例程也是一样的，只是标记不同。

- 使用无赋值的my来创建一个空数组或散列。
- 按空白符分解默认字符串。
- 对变量列表赋值来收集你想要的任意数据。
- 使用未定义引用的自动生成来创建这些引用。
- 对未定义数组和散列元素自增来创建这些元素。
- 使用%seen散列的自增来确定唯一性。
- 对条件中的临时my变量赋值。
- 使用大括号的自动加引号行为。
- 使用一种候选引号机制来内插双引号。

- 使用?:操作符切换printf的两个参数。
- 将printf参数与其%字段对齐:

```
my %seen;
while (<>) {
    my ($a, $b, $c, $d) = split;
    print unless $seen{$a}{$b}{$c}{$d}++;
}
if (my $tmp = $seen{fee}{fie}{foe}{foo}) {
    printf qq(Saw "fee fie foe foo" [sic] %d time%s.\n"),
        $tmp, $tmp == 1 ? "" : "s";
}
```

这9行程序中用到了大量惯用法。第一行建立了一个空散列，因为我们没有为它赋任何元素。这里迭代处理输入行，隐式地设置“它”（即\$\_），然后使用一个无参数split，它会按空白符分解“它”。然后用一个列表赋值选出前4个词，将后面的所有单词舍弃。之后将这前4个单词记在一个4维散列中，这会自动创建（如果必要）前3个引用元素和最后一个用来完成自增的计数元素（在warnings下，自增不会警告你正在使用未定义的值，因为自增是一种可接受的定义未定义值的方法）。如果之前从未见过以这4个单词开头的行，则打印该行。这是因为自增是一个后自增，除了将散列值增1外，它会返回原来的true值（如果有）。

循环之后，我们再次检查%seen来查看是否见过这4个单词的某种特定组合。我们利用了这样一个事实：即可以把直接量标识符放在大括号里，它会自动加引号。否则，我们必须写为\$seen{"fee"}{"fie"}{"foe"}{"foo"}，真是够麻烦的，即使你不是急着从巨人身边逃脱也会觉得它很碍事。

在if提供的布尔上下文中测试\$seen{fee}{fie}{foe}{foo}之前，我们将它的结果赋至一个临时变量。由于赋值返回其左值，我们仍然可以测试这个值来看它是否为true。my告诉你这是一个新变量，我们不是在测试相等性，而是在完成一个赋值。即使没有my它也能很好地工作，专家级Perl程序员仍会立即注意到我们使用了一个=而不是两个(==)。不过，不太熟练的Perl程序可能会糊涂，而对Pascal程序员来说，不论水平如何，这些都会让他们崩溃。

再来看printf语句，可以看到我们使用了qq()形式的双引号，这样也可以内插普通的双引号以及换行符。我们也可以直接在这里内插\$tmp，因为它实际上就是一个双引号字符串，不过我们还是选择通过printf完成进一步的内插。现在临时\$tmp变量很有用，特别是我们不只是想内插\$tmp，而且要在?:操作符的条件中查看是否要对单词“time”加复数。最后，注意我们把两个字段与printf格式中相应的%标记对齐。如果参数太长，无法在一行放下所有参数，可以把下一个参数放在下一行，不过这里不需要这么做。



哇！是不是够多了？还有很多惯用法可以讨论，但这本书已经实在太重了。不过，我们还想再介绍一个Perl惯用法：编写程序生成器。

## 程序生成

几乎从人们最早发现可以写程序开始，他们就开始编写能够写其他程序的程序。我们通常把这些程序称为程序生成器（program generators）。如果你熟悉历史，可能知道RPG在代表角色扮演游戏（Role-Playing Game）很早以前所代表的是报表程序生成器（Report Program Generator）。如今，它们可能被称为“程序工厂”，不过先有了生成器这个名字，所以我们还是用这个名字。

现在任何写过程序生成器的人都知道，即使你很清醒，程序生成器也会让你眼花缭乱。关键的问题在于，程序中的很多数据看起来就像真正的代码，但它们并不是（至少现在还不是）。同一个文本文件中不仅包含完成某些操作的代码，也包括看起来很相似但并不完成任何操作的代码。从文本的角度讲，Perl有很多特性，从而可以很容易把Perl与其他语言混合在一起。

当然，这些特性也使得用Perl编写Perl更为容易，但应该能想到，这不是我们现在要考虑的。

## 用Perl生成其他语言

Perl是一个文本处理语言（当然还有很多其他特性），而且大多数计算机语言都是文本型的。除此以外，Perl没有任意的限制，再加上提供了各种引号和内插机制，这使我们很容易从视觉上把它与所要生成的其他语言代码分开。例如，下面是*s2p*的一小段代码，*s2p*即*sed-to-perl*转换器：

```
print &q(<<"EOT");
:      #!$bin/perl
:      eval 'exec $bin/perl -S \${1+"$@"}'
:          if \${running_under_some_shell};
:
:
EOT
```

这里包围的文本恰好在Perl和*sh*这两个语言中都是合法的。这里使用了一个惯用法，使你在写程序生成器时能保持清醒：我们的技巧是在每个引号行的开头放置一个“噪声”字符和一个tab，这在视觉上可以隔离所包围的代码，所以你一眼就能区分出这不是实际要执行的代码。变量\$bin内插到多行引号字符串中，这里有两处内插，然后把这个字符串传入一个函数，去除其中的冒号和tab。

当然，并不要求一定使用多行引号字符串。人们常会看到CGI脚本包含上百万个print语句，每行一个print语句。这看起来就像是开着Formula 1老式汽车去教堂，不过，说真

的，不知道它能不能开到目的地（必须承认，一组print语句本身在视觉上就有一种区分效果）。

如果嵌入一个很大的多行引号字符串，而且其中包含一些其他语言（如HTML）的代码，可以假装你是从内向外编程，把Perl包围在其他语言中，这样通常很有帮助，PHP等语言就采用这种做法：

```
print <<"XML";
    <stuff>
    <nonsense>
    blah blah blah @{{ scalar EXPR }} blah blah blah
    blah blah blah @{{ LIST }} blah blah blah
    </nonsense>
    </stuff>
XML
```

这两种技巧都可以将任意复杂的表达式的值内插到长字符串中。

有些程序生成器看起来并不像程序生成器，这取决于它们对你隐藏了多少工作。在第19章中，我们见过可以用一个小Makefile.PL程序编写Makefile。得到的Makefile很容易比生成它的Makefile.PL大100倍。想想看这会节省你多少输入，或者干脆不要想。毕竟，这才是关键。

## 用其他语言生成Perl

用Perl很容易生成其他语言，不过反过来也一样。很容易用其他语言生成Perl，因为Perl很简洁，而且具有适应性。你可以挑选引号以免与其他语言的引号机制冲突。不用担心缩进或者换行符要放在哪里，也不用担心如何对反斜线加反斜线。另外，不要求提前将包定义为一个字符串，因为你可以反复进入包的命名空间，只要你想计算这个包中的代码，都可以再次进入这个命名空间。

还有一点有利于用其他语言（包括Perl）写Perl，这就是#line指令。Perl知道如何将它们作为特殊指令处理，来重新配置当前文件名和行号。这在错误或警告消息中很有用，特别是对于用eval处理的字符串（如果仔细想想看，这就是用Perl写Perl）。这种机制所用的正是C预处理器使用的语法：如果Perl遇到一个#符号和单词line，而且后面跟有一个数字和一个文件名，它会把\_\_LINE\_\_设置为这个数，把\_\_FILE\_\_设置为这个文件名<sup>注4</sup>。

这里给出几个例子，可以在perl下直接键入来进行测试。我们使用Control-D指示文件末尾，这是Unix的典型做法。DOS/Windows和VMS用户可以键入Control-Z。如果你的shell

---

注4：从理论上讲，它会匹配模式 `/^#\s*line\s+(\d+)\s*(?:\s"([^\"]+)"?)\s*$/`，\$1提供下一行的行号，\$2提供可选的文件名（在引号里指定）。nul文件名会保持\_\_FILE\_\_不变。

使用其他方式，则要用所要求的那种方式告诉`perl`你已经完成了工作。或者，总是可以输入`_ _END_ _`来告诉编译器没有更多需要解析的内容了。

在这里，Perl的内置`warn`函数会打印出新的文件名和行号：

```
% perl
# line 2000 "Odyssey"
# 上一行中的"#"必须是该行的首字符
warn "pod bay doors"; #或die
^D
pod bay doors at Odyssey line 2001.
```

在这里，`eval`中的`die`产生的异常会放在`$( $EVAL_ERROR )`变量中，另外还有临时新文件名和行：

```
# line 1996 "Odyssey"
eval qq{
#line 2025 "Hal"
    die "pod bay doors";
};
print "Problem with $@";
warn "I'm afraid I can't do that";
^D
Problem with pod bay doors at Hal line 2025.
I'm afraid I can't do that at Odyssey line 2001.
```

这里显示了一个`#line`指令只影响当前编译单元（文件或`eval STRING`），而且这个单元完成编译时，会自动恢复之前的设置。这样一来，你可以在`eval STRING`或`do FILE`中设置你自己的消息，而不会影响程序的其余部分。

`sed-to-perl`转换器（`s2p`）是最早的Perl预处理器之一。实际上，Larry延迟了Perl第一版的发布，就是为了完成`s2p`和`awk-to-perl(a2p)`，因为他认为这两个工具可以提高Perl的接受度。嗯，也许他是对的。

有关的更多信息参见在线文档，以及`find2perl`转换器。

## 源代码过滤器

既然可以编写一个程序将随机的内容转换为Perl，为什么不能想个办法从Perl调用这个转换器呢？

源代码过滤器的概念源于这样一个想法：即一个脚本或模块应当能够动态地自行解密，如下：

```
#!/usr/bin/perl
use MyDecryptFilter;
@*x$]`ouN&k^Zx02jZ^X{. ?s!(f;9Q/^A^@~~8H]|, %@^P:q-=
...
```



不过尽管最初的想法是这样的，但现在源代码过滤器定义为可以对输入文本完成你希望的任何转换。结合第17章提到的`-x`开关的概念，可以得到了一种通用机制，能够从消息中取出任何程序块并执行，而不论它是否用Perl编写。

使用CPAN的Filter模块，现在甚至可以做更有趣的事情，如用`awk`编写Perl：

```
#!/usr/bin/perl
use Filter::exec "a2p"; # awk-to-perl转换器
1,30 { print $1 }
```

也许你认为是这一种惯用法，不过我们会不假思索地认为这是常用实践。

# 可移植的Perl

如果这个世界上只有一个操作系统，可移植会很容易做到，不过生活也就太单调乏味了。我们更愿意有更多不同的操作系统，只要这个生态系统不要划分得太简洁（比如只是区分掠食者和被掠食者而已）。Perl可以在数十种操作系统上运行，因为Perl程序不依赖于平台，所以同一个程序可以在所有这些系统上运行而无需修改。

嗯，只能说几乎无需修改。Perl会尝试为程序员提供尽可能多的特性，不过，如果利用了特定于某个操作系统的特性，肯定就会降低程序对其他系统的可移植性。在这一章中，我们将介绍编写可移植Perl代码的一些原则。一旦确定希望达到何种程度的可移植性，你就能知道在哪里划分界限，并且不超出这个界限。

再从另一个角度来看，编写可移植的代码往往需要限制你的可用选择。很自然地，为此要遵守纪律，而且要做出一定的牺牲，这是Perl程序员可能不太习惯的两个品质。

*perlport*手册页列出了Perl不再支持的一些平台，如Mac OS 9 (Classic) 和Windows 95, 98、ME和NT4。不仅这些平台不再得到支持，原先支持它们的代码也已经从代码库中消失。所以，取决于你使用的Perl，你可能不再需要支持那些代码了。对于Perl支持的系统，如果有调整或者有特殊情况，会在自己的手册页中说明，如表22-1所列。

表22-1：特定于系统的手册页

## 手册页

<i>perlaix</i>	<i>perlfreebsd</i>	<i>perlnetware</i>	<i>perlsymbian</i>
<i>perlamiga</i>	<i>perlhaiku</i>	<i>perlopenbsd</i>	<i>perltru64</i>
<i>perlbeos</i>	<i>perlhpx</i>	<i>perlos2</i>	<i>perluts</i>
<i>perlbs2000</i>	<i>perlhurd</i>	<i>perlos390</i>	<i>perlvmesa</i>

<i>perlce</i>	<i>perlirix</i>	<i>perlos400</i>	<i>perlvms</i>
<i>perlcygwin</i>	<i>perllinux</i>	<i>perlplan9</i>	<i>perlvos</i>
<i>perldgux</i>	<i>perlmacos</i>	<i>perlqnx</i>	<i>perlwin32</i>
<i>perldos</i>	<i>perlmacosx</i>	<i>perlriscos</i>	
<i>perlepoc</i>	<i>perlmpaix</i>	<i>perlsolaris</i>	

并不是所有Perl程序都必须做到可移植。完全可以用Perl把Unix工具“粘合”在一起，或建立一个Macintosh应用的原型，或者可以用来管理Windows注册表。如果有必要牺牲可移植性，那么大可将它舍弃<sup>注1</sup>。一般来讲，需要注意只在多用户平台上才有用户ID、“home”目录甚至记入日志的状态等概念<sup>注2</sup>。

特殊的`$^O`变量会指出你的Perl在哪个操作系统上构建。提供这个信息可以提高代码的速度，否则必须加载Config通过`$Config{osname}`才能得到同样的信息。即使出于另外某种原因加载了Config，也可以利用`$^O`变量从而避免绑定散列查找的开销。还可以使用`Devel::AssertOS`或`Devel::CheckOS` CPAN模块来实现更复杂的控制。

要得到有关平台更详细的信息，可以通过标准Config模块查看%Config散列中的其余信息。例如，要检查平台是否支持`lstat`调用，可以检查`$Config{d_lstat}`。要了解可用变量的完整描述，可以查看Config的在线文档，还可以参见*perlport*手册页，其中给出了一个列表，列出了不同平台上Perl内置函数的行为。下面是在不同平台上行为变化最大的一些Perl函数（参见*perlport*来了解更多细节）：

-X（文件测试）、`accept`、`alarm`、`bind`、`binmode`、`chmod`、`chown`、`chroot`、`connect`、`crypt`、`dbmclose`、`dbmopen`、`dump`、`endgrent`、`endhostent`、`endnetent`、`endprotoent`、`endpwent`、`endservent`、`exec`、`fcntl`、`fileno`、`flock`、`fork`、`getgrent`、`getgrgid`、`getgrnam`、`gethostbyaddr`、`gethostbyname`、`gethostent`、`getlogin`、`getnetbyaddr`、`getnetbyname`、`getnetent`、`getpeername`、`getpgrp`、`getppid`、`getpriority`、`getprotobyname`、`getprotobynumber`、`getprotoent`、`getpwent`、`getpwnam`、`getpwuid`、`getservbyport`、`getservent`、`getservbyname`、`getsockname`、`getsockopt`、`glob`、`ioctl`、`kill`、`link`、`listen`、`lstat`、`msgctl`、`msgget`、`msgrcv`、`msgsnd`、`open`、`pipe`、`qx`、`readlink`、`readpipe`、`recv`、`select`、

注1：并不是多元文化环境中的每一次交谈都必须保证正确。Perl会努力为你至少提供一种正确的方法来完成工作，但它并没有严格要求你必须采用这种方法。在这个方面，Perl更像你的妈妈，而不是保姆。

注2：不过现在“用户”算是一个奇怪的概念，因为即使系统只是为一个人设计的，它也可以有多个“用户”。



semctl、semget、semop、send、sethostent、setgrent、setnetent、setpgrp、setpriority、setprotoent、setpwent、setservent、setsockopt、shmctl、shmget、shmread、shmwrite、shutdown、socket、socketpair、stat、symlink、syscall、sysopen、system、times、truncate、umask、utime、wait、waitpid。

## 换行符

在大多数操作系统上，文件中的行用一个或两个字符终止，这些字符表示行结束。不同系统上，这些结束字符也有所不同。Unix仍采用传统方式使用`\012`（也就是ASCII中的八进制12字符），一类DOS的I/O使用`\015\012`，Unix前的Macs原来使用`\015`。Perl使用`\n`表示一个“逻辑”换行，而不论具体平台是什么。在类DOS的Perl中，`\n`通常表示`\012`，不过以“文本模式”访问一个文件时，它会转换为`\015\012`或从`\015\012`转换回`\012`（这取决于读还是写）。对于采用规范模式的终端，Unix的做法相同。`\015\012`通常表示为CRLF。

由于DOS区分文本文件和二进制文件，所以在“文本模式”的文本上使用`seek`和`tell`时，类DOS的Perl会有一些限制。要得到最好的结果，应当使用`seek`定位到由`tell`得到的位置。不过，如果对文件句柄使用Perl的内置`binmode`函数，那么可以没有任何顾虑地使用`seek`和`tell`。

关于套接字编程有一个常见的误解，认为无论在哪里`\n`都是`\012`。很多常用的互联网协议中指定了`\012`和`\015`，Perl的`\n`和`\r`值并不可靠，因为它们会随系统不同而不同：

```
print SOCKET "Hi there, client!\015\012";      # 正确
print SOCKET "Hi there, client!\r\n";          # 不正确
```

不过，使用`\015\012`（或`\cM\cJ`，或`\x0D\x0A`，或者甚至`v13.10`）可能很麻烦，也不好看，另外还会让维护代码的人感到困惑。Socket模块为需要换行符的人提供了他们想要的东西：

```
use Socket qw(:DEFAULT :crlf);
print SOCKET "Hi there, client!$CRLF"          # 正确
```

从一个套接字读时，要记住默认输入记录分隔符`$/`是`\n`，这表示如果你不能确定通过这个套接字能够看到什么，就必须多做一些工作。健壮的套接字代码应当既能识别`\012`又能识别`\015\012`作为行末字符：

```
use Socket qw(:DEFAULT :crlf);
local ($/) = LF;      # 如果$/已经为\012则不需要

while (<SOCKET>) {
    s/$CR?$LF/\n/;    # 将LF或CRLF替换为逻辑换行符
}
```

类似地，返回文本数据的代码（如获取一个Web页面的子例程）都应当能转换换行符。通常一行代码就足够了：

```
$data =~ s/\015?\012/\n/g;
return $data;
```

## 字节顺序和数字宽度

计算机采用不同的字节顺序（大端顺序*big-endian*或小端顺序*little-endian*）以及不同的宽度（如今最常见的是32位和64位）来存储整数和浮点数。正常情况下不用考虑这个问题。不过，如果你的程序会通过网络连接发送二进制数据，或者要发送到磁盘上由另一个计算机读取，就需要采取一些预防措施了。

如果顺序发生冲突，这可能会导致数字完全错乱。如果一个小端主机（如Intel CPU）存储了0x12345678（十进制的305419896），大端主机（如Motorola CPU）会把它读作0x78563412（十进制为2018915346）。为了避免网络（套接字）连接中出现这个问题，应当使用pack和unpack格式n和N，这会按大端顺序（也称为“网络”顺序）写无符号短整和长整数（而不论平台是什么）。

可以对用原生格式打包的数据结构解包，来研究你的平台采用大端顺序还是小端顺序，例如：

```
say unpack("h*", pack("s2", 1, 2));
# 例如在Intel x86或Alpha 21064等采用小端模式的平台上是'10002000'
# 而在Motorola 68040上是'00100020'
```

要确定字节顺序，可以使用以下语句：

```
$is_big_endian = unpack("h*", pack("s", 1)) =~ /01/;
$is_little_endian = unpack("h*", pack("s", 1)) =~ /^1/;
```

即使这两个系统都采用相同的字节顺序，在32位和64位平台之间传输数据时仍有可能出现问题。对此除了避免传输或存储原始二进制数字外，并没有很好的解决方案。可以采用文本而不是二进制格式传输和存储数字，或者使用类似Data::Dumper或Storable的模块来完成这个工作。实际上，只需要在任何情况下都使用面向文本的协议，它们比二进制协议更健壮、更可维护，也更可扩展。

当然，随着XML和Unicode的出现，我们对文本的定义也变得更加灵活。例如，在两个运行Perl v5.6或更新版本的系统上，可以将整数序列编码为utf8格式的字符来传输（utf8是Perl的UTF-8版本）。如果传输数据的发送端和接收端都运行在支持64位整数的体系结构上，则可以交换64位整数。否则，仅限于32位整数。要用pack结合一个U\*模板来发送数据，另外使用unpack并结合一个U\*模板接收数据（参见第26章）。



# 文件和文件系统

在Unix上，文件路径的各个部分用/分隔，在Windows上用\分隔，而在Unix前的Macs系统上会用:分隔。有些系统既不支持硬链接（link），也不支持符号链接（symlink、readlink、lstat）。有些系统强调文件名的大小写，有些则不区分。另外有些系统关注何时创建文件而不在意何时读文件。而且不同的系统会使用不同的字符集。

要编写管理文件的可移植Perl程序，下面给出有关的一些技巧：

- File::Basename模块是随Perl捆绑发布的另一个平台无关模块，可以将路径名分解为各个部分：基文件名、目录的完整路径以及文件后缀：

```
use File::Basename;

my $name = basename( $ARGV[0] );
my $dir = dirname( $ARGV[0] );

my( $base, $dir, $suffix ) = fileparse( $ARGV[0], qr/\.[^.]+\z/ );
```

- 标准File::Spec模块提供了一些函数，可以在文件系统中移动，另外可以将路径的各个部分正确地重新组合在一起。不要硬编码设置路径，而应动态构建：

```
use File::Spec::Functions;
chdir( updir() );      # 上移一层目录
$file = catfile( curdir(), "temp", "file.txt" );
```

在Unix和Windows上，最后一行会读入./temp/file.txt，或者在VMS上会读入[.temp]file.txt，并把该文件的内容存储在\$file中。

- CPAN的File::HomeDir模块可以检测操作系统，并构造正确的路径从而定位特殊的用户目录。
- 使用Path::Class CPAN模块得到File::Spec的面向对象接口，从而可以方便地从一种系统读取路径，并将它转换为另一个系统的一个等价路径。
- 使用File::Temp模块（Perl提供）创建一个临时文件（或者目前为止该文件的名称尚未用过）。
- 不要将同名的两个文件用于不同的情况，如test.pl和Test.pl，因为一些平台会忽略首字母大写（有些会忽略大小写，但仍保留原形式）。
- 限制文件名尽可能遵循8.3约定（8个字母的文件名，3个字母的扩展名）。只要遵循这种8.3约定（嘿，这可比让骆驼穿过针眼简单多了）时能确保文件名唯一，通常可以避免使用更长的文件名。
- 尽量减少文件名中的非字母数字字符。使用下划线通常是可以的，不过这会浪费一个字符，否则在遵循8.3约定的情况下多一个字符可以更好地保证唯一性（要记住，这也是为什么我们不在模块名中放入下划线的原因）。



- 规范化文件名，或者避免使用非ASCII字符。不同系统上对Unicode文件名的支持是不同的，而且没有一个适用于所有系统的通用API。有些字符在一些系统上可用，但在另外一些系统上完全不可用。
- 类似的，使用AutoSplit模块时，要尽量限制子例程名为8个字符或少于8个字符，另外，不要提供两个同名的子例程用于不同的情况。如果需要更长的子例程名，要让每个子例程名的前8个字符是唯一的。
- 总是使用<显式打开用于读取的文件。否则，在允许文件名中包含标点符号的系统上，有>字符作为前缀的文件名会导致文件被清除，而有|字符前缀的文件名会导致一个管道式open。这是因为两参数形式的open是有魔法的，它会解释类似>、<和|的字符，这可能不是你的本意（除非你有意如此）。

```
open(FILE,      $existing_file) || die $!; # 不正确
open(FILE,      "<$existing_file") || die $!; # 正确一些
open(FILE, "<", $existing_file) || die $!; # 更正确
```

- 选择输入和输出编码，并建立相应文档来提供说明。如果能为人们提供一种方法来选择他们想要的编码则更好。如果你不知道想要什么编码，可以使用UTF-8。要避免UTF-16，它存在很多字节顺序问题。
- 不要假设文本文件会以一个换行符结束。尽管本该如此，但有时人们会忘记加换行符，特别是由于有文本编辑器的帮助，使人们更容易忘记这一点。

## 系统交互

依赖于图形用户界面的平台有时缺少命令行工具，所以需要命令行接口的程序可能并非在任何地方都能运行。对此，除了升级往往别无他法。

下面再给出一些技巧：

- 有些平台不会删除或重命名正在使用的文件，所以用完文件时一定要记得将它关闭。不要对一个打开的文件取消链接（unlink）或重命名（rename）。不要绑定（tie）或打开（open）一个已经绑定或打开的文件，要先用untie或close将它解除绑定或关闭。
- 不要将同一个文件打开多次来完成写操作，因为一些操作系统会对这种文件加强制锁。
- 不要依赖于%ENV中的一个特定的环境变量，另外不要假设%ENV中的所有变量都区分大小写或保留大小写。不要假设环境变量采用Unix继承语义。在一些系统上，它们可能对所有其他进程都可见。
- 不要使用信号或%SIG。

- 避免使用文件名匹配 (globbing)，而应当使用`opendir`、`readdir`和`closedir`（在v5.6中，基本文件名匹配的可移植性比原先要好，不过如果你想搞点花样，有些系统在类Unix的默认接口下仍会失败）。
- 不要假设错误号或错误字符串的特定值存储在`$!`中。

## 进程间通信 (IPC)

为了保证最大的可移植性，不要试图启动新进程。这意味着要避免`system`、`exec`、`fork`、`pipe`、```、`qx//`或带`|`的`open`。

主要问题不在于操作符本身。通常大多数平台上都支持启动外部进程的命令（不过有些平台上不支持任何类型的新进程派生）。如果你要调用外部程序，这些程序的名字、位置、输出或参数语义在不同平台上可能不同，这种情况下就会有问题。

下面是一个非常流行的Perl代码，它会打开一个到`sendmail`的管道，使程序能够发送邮件：

```
open(MAIL, "|/usr/lib/sendmail -t") || die "cannot fork sendmail: $!";
```

在没有`sendmail`的平台上这将无法工作。要得到一个可移植的解决方案，可以使用某个CPAN模块来发送邮件，如MailTools发行版中的`Mail::Mailer`和`Mail::Send`模块，或者`Mail::Sendmail`。

Unix System V IPC函数 (`msg*`()、`sem*`()、`shm*`()) 并不总是可用的，甚至在一些Unix平台上也有可能不可用。

`IPC::Run`、`IPC::System::Simple`和`Capture::Tiny` CPAN模块可以帮助处理一些外部命令的跨平台问题。

## 外部子例程 (XS)

通常XS代码可以用于任何平台，不过库和头文件不一定可用，或者XS代码本身就可能是特定于平台的。如果库和头文件可移植，那么有理由认为XS代码也可以移植。

写XS代码时，可能出现另外一类可移植问题：即最终用户平台上是否有可用的C编译器。C会引入它自己的可移植性问题，而编写XS代码就可能遭遇这样一些问题。用纯Perl编写代码可以更容易地得到可移植性，因为Perl的配置过程做了很多工作，可以为你隐藏C的可移植性问题<sup>注3</sup>。

---

注3： 有些处于社会边缘的人会运行Perl的`Configure`脚本，把它当作一种廉价的消遣方式。很多人据说还在竞争的系统间设立了“`Configure`竞赛”，并投下大把赌注。如今这种做法在大多数文明世界里都认为是不合法的。



## 标准模块

一般来讲，标准模块（与Perl捆绑发布的模块）在所有平台上都可以运行。但有几个例外需要注意：CPAN.pm模块（目前与可能不可用的外部程序连接）、平台特定的模块（如ExtUtils::MM\_VMS）和DBM模块。

不是所有平台上都有一个DBM模块。通常所有Unix和类DOS的移植版本中都有SDBM\_File和其他相应模块。

好的一点是，至少会有一个DBM模块，而且AnyDBM\_File会用它找到的任何模块。由于有这种不确定性，应当只使用所有DBM实现共有的特性。例如，保证记录不超过1K字节。有关的更多详细信息参见AnyDBM\_File模块文档。

比DBM更复杂的是SQLite，它由面向DBI的DBD::SQLite驱动程序提供。这是公共域中一个最小的可内嵌关系数据库（所以可以随代码发布）。它在常用操作系统上都能运行。

## 日期与时间

尽可能使用ISO-8601标准（“YYYY-MM-DD”）来表示日期。通过使用Date::Parse等模块，类似“1987-12-18”的字符串可以很容易地转换为系统特定的一个值。使用Time::Local可以将一组时间和日期值（如内置localtime函数返回的值）转换为系统特定的表示。

内置time函数总会返回自“纪元”开始以来的秒数，不过操作系统对于“纪元”时间的理解不同。在很多系统上，纪元从1970年1月1日00:00:00 UTC开始，不过在VMS上，这是从1858年11月17日00:00:00开始的。所以，为了得到可移植的时间，往往要计算纪元的偏移量：

```
require Time::Local;
$offset = Time::Local::timegm(0, 0, 0, 1, 0, 70);
```

Unix和Windows上\$offset的值总是0，不过在其他系统上这可能是一个很大的数。然后再把\$offset增加到一个Unix时间值，这样可以得到任何系统上都相等的一个值。

可以用很多不同方法来控制系统的当日时间和日历日期的表示。不要假设时区存储在\$ENV{TZ}中，即使存储在这里，也不要假设你能通过这个变量控制时区。

如果需要特别精确的日期和时间控制，可以利用CPAN的DateTime模块。

## 国际化

不要对编码或环境做任何假设。也许你和你认识的所有人都使用相同的设置，但是一旦发布你的程序，就会看到一个充满了差别的世界。



在程序中应当使用Unicode。要在通向外部世界的接口中完成与其他字符集转换。参见第6章。

在Unicode世界之外，对于字符集要做尽可能少的假设，而且对字符的ord值不能做任何假设。不要假设字母字符一定有顺序ord值。小写字母可能在大写字母前面，也可能在大写字母后面。小写和大写的顺序可能会前后交错，如a和A都在b前面。重音符和其他国际字符也可能是前后交错的，如ä在b前面。

即使使用Unicode，这些警告依然成立。很多字母字符序列的码点与其字母顺序没有任何关系。

如果你的程序要在一个POSIX系统上运行（一个很大的假设），可以参考*perllocale*手册页得到有关POSIX本地化环境的更多信息。本地化环境会影响字符集和编码以及日期和时间格式，另外还会影响很多其他方面。适当地使用本地化环境会让你的程序更可移植，至少对于非英语用户来说会更方便，更友好。不过，要当心，本地化环境和Unicode还不能很好地结合。

## 风格

必须编写特定于平台的代码时，可以考虑把它们放在一个地方，以便移植到其他平台。要使用Config模块和特殊变量`$^O`来区分不同的平台。

对于随模块或程序提供的测试，一定要当心。也许一个模块的代码是完全可移植的，但是它的测试可能并不能很好地移植。如果要派生其他进程或调用外部程序来辅助测试，或者如果（如前所述）测试对文件系统和路径做了某些假设，通常就会发生这种情况。要注意不要依赖特定的输出风格来检查错误，甚至在系统调用后检查`$!`来查找“标准”错误时也是如此。检查错误应当使用Errno模块。

要记住，好的风格是超越时间和文化的，所以，为了得到最大的可移植性，必须努力了解你所在的这个宇宙。最酷的人不是那些一味追赶时尚的人，他们不一定最酷，因为他们没有考虑如何“保持”他们自己的文化（不论是编写程序还是其他方面）。时尚是变化的，但风格永远不变。

# Pod

Perl设计的基本原则之一是让简单的工作很容易，让困难的工作也并非遥不可及。文档就应该简单。

Perl支持一种简单的文本标记格式，称为`pod`，它可以独立存在，也可以与源代码自由地结合来创建内嵌文档。Pod可以转换为很多其他格式来进行打印或浏览，或者你也可以直接阅读，因为它是无格式的。

Pod的表现力不如XML、 $\text{\LaTeX}$ 、*troff*(1)等语言，甚至比不上HTML。我们是有意这样设计的：通过牺牲表现力来换取简单性和方便性。有些文本标记语言则不然，它们要求作者写的标记比文本还要多，这就会不必要地增加写文档的难度，而且阅读这种文本几乎是不可能的。一个好的格式应该就像优秀的电影配乐一样，要隐藏在后台而不应让人分心。

让程序员写文档几乎与让他们扎领带一样困难。所以我们特意将Pod设计得非常容易，使得甚至程序员也能写文档，而且愿意写文档。并不是说能用pod来写一本书，不过，写这本书它确实是完全胜任的。

## Pod核心技术

大多数文档格式要求整个文档都采用那种格式。Pod则比较宽容：你可以把pod嵌入到任何类型的文件中，然后依赖于pod转换器提取pod。有些文件100%由纯pod组成。不过另外一些文件（特别是Perl程序和模块）可能包含少量pod，它们分散在程序中作者认为合适的地方。解析文件准备执行时，Perl会跳过这些pod文本。

在本该是语句的位置上，如果Perl词法分析器看到的文本行以一个等号和一个标识符开头，它就会知道从这里开始是文档，需要跳过这一部分，如下所示：

```
=head1 Here There Be Pods!
```

从这个文本开始，直到后面以=cut开头的文本行（包括=cut行），所有这些文本都将被忽略。这就允许你自由地混合源代码和文档，如下所示：

```
=item snazzle
```

```
The snazzle() function will behave in the most spectacular
form that you can possibly imagine, not even excepting
cybernetic pyrotechnics.
```

```
=cut
```

```
sub snazzle {
    my $arg = shift;
    ....
}
```

```
=item razzle
```

```
The razzle() function enables autodidactic epistemology generation.
```

```
=cut
```

```
sub razzle {
    print "Epistemology generation unimplemented on this platform.\n";
}
```

要了解更多有关的例子，可以参见标准或CPAN Perl模块。它们都应该提供pod，而且几乎所有模块都确实提供了pod，也有少数例外。

由于Perl语法分析器会识别pod并将其丢掉，你还可以使用一个适当的pod指令将大段代码快速注释掉。使用=for pod块可以注释掉一个段落（paragraph），或者可以使用一个=begin/=end对将更大的节（section）注释掉。后面还会介绍这些pod指令的语法。不过要记住，对于这两种情况，即在=for pod块或=begin/=end对之后仍在pod模式下，所以需要再用=cut返回编译器：

```
print "got 1\n";
```

```
=for commentary
```

```
This paragraph alone is ignored by anyone except the
mythical "commentary" translator. When it's over, you're
still in pod mode, not program mode.
```

```
print "got 2\n";
```

```
=cut
```

```
# ok, real program again
```

```
print "got 3\n";
```

```
=begin comment
```

```
print "got 4\n";
```

```
all of this stuff
```



```

    here will be ignored
    by everyone

    print "got 5\n";

    =end comment

    =cut

    print "got 6\n";

```

这会打印得到了1、3和6。要记住，并不是任何地方都能放这些pod指令。解析器希望看到一个新语句的地方才能放pod指令，而不能把pod指令放在一个表达式的中间或者其他任意的位置上。

从Perl的角度来看，会把所有pod标记丢掉。不过，从pod转换器的角度看，丢掉的是代码。Pod转换器把其余的文本都看作是用空行分隔的一系列段落。

目前共有3种段落：字面段落（verbatim paragraphs）、命令段落（command paragraphs）和散文段落（prose paragraphs）。

## 字面段落

字面段落用于你希望原样显示的字面文本，如代码段。字面段落必须缩进。也就是说，它必须以一个空格或tab字符开头。转换器会原样重新生成这些文本，通常采用一种等宽字体，一般认为tab宽度为8列。这里没有特殊的格式化转义，所以不能把字体变成斜体或加粗。<字符表示一个直接量<，没有任何其他含义。

## 命令段落

所有pod指令开头都有一个=，后面跟有一个标识符。接下来可能有任意数量的文本（指令可以采用它喜欢的任何方式使用这些文本）。这里唯一的语法要求是文本必须在一个段落中。目前可以识别的指令（有时称为pod命令）包括：

**=encoding**

默认地，Pod转换器假设Pod源编码为ASCII或Latin-1。可以用这个命令指定编码来改变默认设置，可以指定为UTF-8：

```
=encoding uft8
```

**=head1**

**=head2**

=head1、=head2等指令会生成指定级别的标题。段落中的其余文本会处理为标题描述。这类似于man(7)中的.SH和.SS节和子节标题，或者HTML中的<H1>…</H1>和<H2>…</H2>标记。实际上，对于这些格式（man(7)或HTML），转换器正是要把

=head1、=head2等指令转换为这些标题或标记。

=cut

=cut指令指示到达pod末尾（文档后面可能还有其他pod，不过，如果是这样，那些pod会由另外的pod指令引入）。

=pod

=pod指令只是告诉编译器不要解析从这里开始一直到下一个=cut的代码，除此以外它什么也不做。如果要大量混合代码和pod，可以用它为文档增加另外一个段落。

=over *NUMBER*

=item *SYMBOL*

=back

=over指令开始一节，专门用于使用=item指令生成列表。在列表的最后，要用=back结束列表。如果提供了*NUMBER*，这会暗示格式化工具需要缩进多少个空格。有些格式化工具还不够丰富，不能识别这些暗示，不过另外还有一些格式化工具又过于丰富，以至于处理比例字体时很难只通过数空格来完成对齐，所以也无法支持这些暗示（不过，通常4个空格已经为项目符号或编号留出了足够的空间）。

列表的具体类型由各个项上的*SYMBOL*指示。下面是一个项目符号列表：

```
=over 4
```

```
=item *
```

```
Mithril armor
```

```
=item *
```

```
Elven cloak
```

```
=back
```

这里是一个编号列表：

```
=over 4
```

```
=item 1.
```

```
First, speak "friend".
```

```
=item 2.
```

```
Second, enter Moria.
```

```
=back
```

以下是一个命名列表：

```
=over 4
```

```
=item armor()
```

Description of the armor() function

=item chant()

Description of the chant() function

=back

可以嵌套相同或不同类型的列表，不过要遵循一些基本规则：不要在=over/=back块之外使用=item；在=over/=back块中至少要使用一个=item；另外（这可能也是最重要的），要保证给定列表中的各项类型一致。可以对各项使用=item \*来生成一个项目符号列表；或者使用=item 1., =item 2等生成一个编号列表；还可以使用=item foo, =item bar等生成一个命名列表。如果开始是项目符号或编号，后面也应当使用同一类型的项，因为格式化工具可以使用第一个=item类型来确定如何格式化这个列表。

与pod的其他方面一样，结果取决于转换器。有些转换器会注意跟在=item后面的特定数字（或字母，或者罗马数字），但另外一些转换器不会注意这些。例如，当前使用的pod2html转换器就很“散漫”：它会完全去除序列指示符（而不是查看这些指示符来推断你使用的是哪一种序列），然后把整个列表都包围在<OL>和</OL>标记中，这样一来，浏览器可以把它显示为一个HTML有序列表。不能认为这是一个特性，将来这个问题可能会得到修正。

=for TRANSLATOR

=begin TRANSLATOR

=end TRANSLATOR

=for、=begin和=end允许包含特殊的节（section），这些节可以不做任何修改地传递，不过只传递到某些特定的格式化工具。如果格式化工具识别出TRANSLATOR中是自己的名字或别名，就会关注这些指令；而所有其他格式化工具则完全忽略这些指令。指令=for指定这个段落的其余部分要传递到一个特定的转换器：

=for html

<p> This is a<flash>raw</flash> <small>HTML</small> paragraph </p>

=begin和=end指令对的工作类似于=for，不过，它们不只接收一个段落，而会把匹配的=begin和=end之间的所有文本都传递到一个特定的转换器。下面给出几个例子：

=begin html

<br> Figure 1.<IMG SRC="figure1.png"><br>

=end html

=begin text

-----  
| foo |



```
      |   bar   |  
-----  
^^^^ Figure 1.^^^^  
  
=end text
```

格式化工具接受的`TRANSLATOR`值通常包括`roff`、`man`、`troff`、`nroff`、`tbl`、`eqn`、`latex`、`tex`、`html`和`text`。有些格式化工具会接受其中一些值作为同义词。所有转换器都不会接受`comment`，这只是一个习惯用语，用来表示将被忽略的内容。无法识别的词也有同样的作用。写这本书时，我们通常用指令`=for later`留下一些记录<sup>注1</sup>。

需要说明，`=begin`和`=end`确实可以嵌套，不过前提是最外层匹配的`=begin`和`=end`指令将使中间的所有内容都处理为非pod（即使其中恰好包含其他`=word`指令）。也就是说，一旦任何转换器看到`=begin foo`，它就会忽略或处理直到相应`=end foo`的所有内容。

## 流文本

第三类段落是“流”文本。也就是说，如果一个段落不是以空白符或一个等号开头，会把它当作一个无格式段落：也就是通常键入的常规文本（修饰尽可能少）。会把换行符处理为等价于空格。主要由转换器负责美观问题，因为程序员还有更重要的事情要做。一般认为转换器会应用某些常用的提示，参见这一章后面的“pod转换器和模块”。

不过，可以明确地做一些事情。在普通段落或标题/项指令中（不过不包括字面段落），可以使用特殊的序列来调整格式。这些序列总是以一个大写字母加一个左尖括号开头，并一直延伸到（不一定是下一个）右尖括号。序列中还可以包含其他序列。

下面是pod定义的一些序列：

### **I<text>**

斜体文本，用于强调、书名、船名和手册页引用如“*perlpod(1)*”。

### **B<text>**

加粗文本，几乎只用于命令行开关，有时也用于程序名。

### **C<text>**

字面代码，可能采用一种定宽字体，如Courier。某些将作为代码的简单项可能不需要这个序列（转换器应该能推导出它们将作为代码），不过最好还是应该加上。

---

注1：实际上我们创建了自己的pod转换器，使用Pod::PseudoPod的一个定制子类将pod源代码转换为DocBook。

**S<text>**

包含非换行空格的文本。通常用于包含其他序列。

**L<name>**

一个名字的交叉引用（链接）：

**L<name>**

手册页。

**L<name/ident>**

手册页中的项。

**L<name/"sec">**

另一个手册页中的节。

**L</"sec">**

同上。

接下来5个序列与上面的5个序列相同，不过只输出text，链接信息会像HTML中一样隐藏：

**L<text/name>**

**L<text/name/ident>**

**L<text/name/"sec">**

**L<text/"sec">**

**L<text/"sec">**

text不能包含字符“/”和“|”，而且应当包含匹配的“<”和“>”。

**F<pathname>**

用于文件名。据说这等同于I。

**X<entry>**

某种索引项。与以往一样，具体怎么做由转换器决定。Pod规范对此没有说明。

**E<escape>**

一个命名字符，类似于HTML转义：

**E<lt>**

直接量<（除非在其他内部序列中而且前面是一个大写字母，否则是可选的）。

**E<gt>**

直接量>（除非在其他内部序列中，否则是可选的）。

**E<sol>**

直接量/（仅L<>中需要）。

E<verbar>

直接量|（仅L<>中需要）。

E<NNN>

E<0XXXXXX>

字符编号（即码点）NNN或Unicode码0XXXXXX。

E<entity>

一些非数字HTML实体，如E<Agrave>。

Z<>

一个0宽度字符。如果某个序列可能带来混淆，在它前面加上Z<>序列就很合适。例如，如果在常规的散文段落中有一行必须以一个等号开头，就可以写为：

Z<>=can you see

或者如果某一行包含一个“From”，也可以加上Z<>序列，这样邮件程序就不会再增加一个大于号：

Z<>From here on out...

大部分情况下，需要一组尖括号来为pod序列定界。不过，有时可能希望在一个序列内部放入一个<或>（使用C<>序列为一个代码段提供等宽字体时这种情况尤其常见）。类似于Perl的所有其他方面：条条大路通罗马。一种简单的办法是用一个E<ENTITY>序列表示结束尖括号：

C<\$a E<lt>=E<gt> \$b>

这会生成“\$a <=> \$b”。

一种更可读（可能也更“朴素”）的方法是使用一组不要求对尖括号转义的候选定界符。可以使用双尖括号（C<< stuff >>），前提是空白符紧跟在开始定界符后面，而且后面紧跟着结束定界符。例如，可以像下面这样：

C<< \$a <=> \$b >>

可以根据需要使用多个重复的尖括号，只要开始和结束尖括号数相同。另外，必须确保空白符紧跟在左边最后一个<后面，而且后面紧跟着右边的第一个>。所以下面这样是可以的：

C<<< \$a <=> \$b >>>

C<<<< \$a <=> \$b >>>>

所有这些最后都会生成采用一种等宽字体的\$a <=> \$b。

两边额外的空白符都会消失，所以如果你愿意，要保留外面的空白符。另外，内部



的两组额外空白符不会重叠，所以，如果引用的第一个东西是>>，不会把它当作结束定界符：

```
The C<< >> >> right shift operator.
```

这会生成“The >> right shift operator.”。

需要说明，pod序列确实可以嵌套。这说明你可以写“The I<Santa MarE<iacute>a> left port already”来生成“The *Santa María* left port already”，或者写“B<touch> S<B<-t> I<time>> I<file>”来生成“*touch -t time file*”，这些都能正确工作。

## Pod转换器和模块

Perl捆绑发布了多个pod转换器，可以把pod文档（或内嵌在其他类型文档中的pod）转换为不同格式。所有这些转换器都应当可以处理8位编码。

### pod2text

将pod转换为文本。正常情况下，这个文本应当是7位ASCII，但是如果它有8位输入，则会有8位，或者如果你使用类似LE<uacute>thien的序列表示*Lúthien*，或使用EE<auml>rendil序列表示*Eärendil*，则采用ISO-8859-1（或Unicode）。

如果文件中包含pod，只查看格式化的pod时，最容易的方法是（不过可能并不是最漂亮的方法）：

```
% pod2text File.pm | more
```

再次说明，一般认为pod是人可读的而不需要格式化。

### pod2man

将pod转换为Unix手册页格式，从而适合通过*nroff*(1)查看，或者通过*troff*(1)创建排版副本。例如：

```
% pod2man File.pm | nroff -man | more
```

或者：

```
% pod2man File.pm | troff -man -Tps -t > tmppage.ps
% ghostview tmppage.ps
```

要打印：

```
% lpr -Ppostscript tmppage.ps
```

### pod2html

将pod转换为HTML，从而可以用你喜欢的浏览器（可能是*lynx*）查看：

```
% pod2man File.pm | troff -man -Tps -t > tmppage.ps
% lynx tmppage.html
```

*pod2latex*

将pod转换为L<sup>A</sup>T<sub>E</sub>X格式，然后可以用这个工具排版。

CPAN还提供了针对其他格式的转换器。

写pod时应当尽可能接近无格式文本，要用尽可能少的显式标记。应当由各个转换器来决定如何表现文本中的内容。这说明，要让转换器确定如何创建成对的引号，如何填充和调整文本，如何为全大写的单词找到一种更小的字体等。由于这些要用来处理Perl文档，大多数转换器<sup>注2</sup>还应当能够识别类似下面这些无修饰的项，并适当地显示：

- FILEHANDLE
- \$scalar
- @array
- function()
- somebody@someplace.com
- http://foo.com/

Perl还提供了多个标准模块来解析和转换pod，包括Pod::Checker（和相关的*podchecker*工具）用于检查pod文档的语法，Pod::Find用于在目录树中查找pod文档，以及Pod::Simple用于创建你自己的pod工具。在一个CPAN发行版本中，可以使用Test::Pod模块检查文档的格式，并使用Test::Pod::Coverage模块检查所有接口是否都有相应的文档。

需要说明，pod转换器应当只查看以一个pod指令开头的段落（从而使解析更为容易），而编译器知道要查找pos转义，即使它在一个段落的中间。这说明，编译器和转换器都会忽略下面的秘密内容。

```
$a=3;
=secret stuff
warn "Neither POD nor CODE!?"
=cut back
print "got $a\n";
```

永远也不要依赖于pod生成的warn。并不是所有pod转换器在这方面都有很好的表现，将来某一天编译器可能会变得更挑剔。

## 编写自己的Pod工具

Pod设计的初衷就是要容易写。pod的简单性还有一个附加的好处，编写简单的工具来处

---

注2： 如果你要设计一个不只是面向Perl代码的通用pod转换器，你的规则可能有所不同。

理pod也很容易。如果想查找pod指令，只需要设置输入记录分隔符为段落模式（可能要用`-00`开关），只关注那些看起来像pod的段落。

例如，下面是一个简单的`olpod`程序，可以用来生成pod大纲：

```
#!/usr/bin/perl -l00n
# olpod - outline pod
next unless /^=head/;
s/^=head(\d)\s+/ " " x ($1 * 4 - 4)/e;
print $_, "\n";
```

如果对本书这一章运行这个程序，会得到类似这样的结果：

```
POD
Pod核心技术
    字面段落
    命令段落
    流文本
Pod转换器和模块
编写自己的Pod工具
Pod陷阱
为Perl程序建立文档
```

这个pod大纲生成器并不关心是否在一个合法的pod块中。因为pod和非pod可以混合在同一个文件中，运行通用工具来搜索或分析整个文件不一定有意义。不过这也是可以做到的，因为为pod写工具非常容易。下面给出一个工具，它了解pod与非pod之间的差别，而且只生成pod：

```
#!/usr/bin/perl -00
# catpod - cat out just the pods
while (<>) {
    if (! $inpod) { $inpod = /^=/ }
    if ($inpod) { $inpod = !/^=cut/; print }
} continue {
    if (eof) { close ARGV; $inpod = "" }
}
```

可以在另一个Perl程序或模块中使用这个程序，然后将输出管道传输到另一个工具。例如，如果有`wc(1)`程序<sup>注3</sup>来统计行数、单词数和字符数，可以为它提供`catpod`输出，统计时只考虑pod部分：

```
% catpod MyModule.pm | wc
```

很多情况下pod允许你使用简单的Perl编写基本工具。既然可以把`catpod`作为一个组件，下面再提供一个工具，只显示缩进的代码：

```
#!/usr/bin/perl -n00
```

---

注3： 如果没有，可以从CPAN找到Perl Power Tools版本。



```
# podlit - print the indented literal blocks from pod input
print if /^\\s/;
```

用它能做什么？嗯，你可能想对文档中的代码完成`perl -wc`检查。或者也许想用某个`grep(1)`<sup>注4</sup>只查看代码示例：

```
% catpod MyModule.pm | podlit | grep funcname
```

这种基于可交换部件（而且可以分开测试）的工具兼过滤器策略是设计可重用软件组件的一种极其简单而强大的方法。这是懒惰的一种形式，只合成一个最小的解决方案来完成当前的工作（至少完成某些类型的任务）。

不过，对于其他任务则可能达不到预期目标。从头开始写一个工具有时需要做更多的工作，有时反而会更少。对于我们之前给出的例子，由于Perl本身具有强大的文本处理能力，所以很适合使用强力处理。不过并非所有情况都是如此。处理pod时可能注意到，尽管pod指令很容易解析，但是pod指令序列可能有些不确定。指令序列可以嵌套在其他序列中（尽管一些不完备的转换器不支持这一点），而且可以有变长的定界符。

并不是所有这些解析代码都由你自己编写，可以从懒惰性出发寻找另一个解决方法。标准Pod::Simple模块可以满足这个需要。这对于复杂的任务尤其有用，如需要解析段落的内部细节，转换为某种候选的输出格式等。使用这个模块可以更容易地完成复杂的任务，因为最终你要写的代码会少得多。这种方法比较好的原因还在于已经为你完成了复杂的解析工作。这与管线中使用catpod的原则实际上是一样的。

Pod::Simple模块采用一种有趣的方法来完成工作。这是一个面向对象的模块，与你在本书中见过的大多数模块都不同。它的主要目标并不是提供可以直接管理的对象，而是提供一个基类，可以在这个基类上构建其他类。

可以创建你自己的类，并继承Pod::Simple（或它的某个接口），Pod::Simple提供了解析pod的所有方法。你的子类要覆盖适当的方法，将解析器找到的结果转换为你想要的输出。应当只覆盖你想改变的部分。可以从一个与你的目标很接近的转换器开始。下面是Pod::Simple::Text的一个小子类，它会查找L<>中的Perl文档实例，并为每个链接建立附注。你必须了解基类的内部结构，这违反了封装原则，不过我们只是为了展示而没有商业目的：

```
use v5.14;

package Local::MyText 0.01 {
    use parent "Pod::Simple::Text";
    use Data::Dumper;
    my @links;
```

---

注4： 如果没有`grep`，请看前一个脚注。

```

sub links {
    $_[0]->{"."_PACKAGE_"}{links} //= [];
}

sub start_L {
    my($self, $link) = @_;
    push $self->links, $link->{to}[2];
}

sub end_L {
    my($self) = @_;
    my $count = @{$self->links};
    $self->{Thispara} .= "[" . $count . "];
}

sub end_Document {
    my($self) = shift;
    while (my($index, $text) = each $self->links) {
        $self->{Thispara} .=
            "$index http://perldoc.perl.org/$text.html\n";
    }
    $self->emit_par;
}
}

1;

```

可以编写你自己的`pod2text`，加载一个文件并调用你的子类，不过`perldoc`可以用`-M`开关<sup>注5</sup>加载一个候选的格式化类：

```
% perldoc -MLocal::MyText some_pod.pod
```

对于以下pod：

```
=pod
```

```
If you want to read about the Perl pod specification, see
the LZ<><perlpod> or LZ<><perlpodspec> documentation.
```

```
=cut
```

可以得到以下输出：

```
If you want to read about the Perl pod specification, see the
perlpod[1] or perlpodspec[2] documentation.
```

```
0 http://perldoc.perl.org/perlpod.html
1 http://perldoc.perl.org/perlpodspec.html
```

这个例子只会改变格式化工具如何解释pod规范。下面给出另一个例子，这里覆盖了字面段落的处理，要用`Perl::Tidy`对它们重新格式化：

---

注5： 这与perl的`-M`开关不是同一个东西。`-M`和类名之间必须有一个空格。很遗憾，即使你弄错了，`perldoc`也不会向你发出警告。

```

use v5.14;

package Local::MyTidy 0.01 {
    use parent "Pod::Simple::Text";
    use Perl::Tidy;

    sub end_Verbatim {
        my($self) = @_;
        Perl::Tidy::perltidy(
            source      => \ $self->{Thispara},
            destination => \ my $out,
            argv        => [qw/-gnu/],
        );
        $self->{Thispara} = $out =~ s/^/ /gmr;
        say { $self->{output_fh} } "", $self->{Thispara};
        return;
    }
}

1;

```

这个格式化工具可以把类似下面的代码（格式很糟糕）：

```

=encoding utf8

=pod

This is a regular paragraph.

#!/usr/bin/perl
for (@ARGV){
    my $count = 0;
    say $count++, " ", $_;
}

This is another regular paragraph.

=cut

```

转换为<sup>注6</sup>：

```

This is a regular paragraph

#!/usr/bin/perl
for (@ARGV) {
    my $count = 0;
    say $count++, " ", $_;
}

This is another regular paragraph

```

还可以定义新的pod命令。例如，如果你想定义一个新命令，这也很容易做到，只是要做一些调整。必须告诉解析器你的新命令是合法的。在下面的例子中，要定义一个新的V<>

---

注6： Perl::Tidy模块还接收很多不同的选项，你可以调整这些选项来选择你喜欢的风格。



命令将它的文本转换为一个码点列表。所以你不会看到é，而将看到（U+00E9）。为此，进入V<>时要设置一个标志，使它知道要在handle\_text中做些不同的处理：

```
use v5.14;
package Local::MyCodePoint 0.01 {
    use parent "Pod::Simple::Text";
    use Data::Dumper;

    sub new {
        my $self = shift;
        my $new = $self->SUPER::new;
        $new->accept_codes("V");
        return $new;
    }
    sub handle_text {
        my($self, $text) = @_;
        $self->{Thispara} .=
            $self->{"."_PACKAGE_"}{in_V}
                ? $self->make_codepoints($text)
                : $text;
    }
    sub make_codepoints {
        $_[1] =~ s/(.)/ sprintf "(U+%04X)", ord($1) /ger;
    }
    sub start_V {
        my($self, $text) = @_;
        $self->{"."_PACKAGE_"}{in_V} = 1;
    }
    sub end_V {
        my($self, $text) = @_;
        $self->{"."_PACKAGE_"}{in_V} = 0;
    }
}

1;
```

利用这个新命令，以下pod：

```
=encoding utf8

=pod

V<À> la recherche du temps perdu

=cut
```

会变成下面的文本：

```
(U+00C0) la recherche du temps perdu
```

## Pod陷阱

Pod很简单，不过也有可能出问题，让一些转换器不知所措：

- 很容易漏掉末尾的尖括号。
- 很容易漏掉末尾的`=back`指令。
- 很容易不小心在一个长的`=for comment`指令中间放入一个空行。可以考虑使用`=begin/=end`来代替。
- 如果`=begin/=end`对中某个标记键入错误，它会吞掉文件的其余部分（把它们当作pod）。可以考虑使用`=for`来代替。
- Pod转换器要求段落用完全的空行分隔，也就是说，要用两个或更多连续的换行（`\n`）字符分隔。如果某行中包含空格或tab，则不会把它当作空行。这可能导致两个或多个段落被当作一个段落。
- “链接”的含义不是由pod定义，而要由各个转换器来确定如何处理（如果你从一开始就认为大多数决定都要由转换器做出，而不是由pod决定，那么你是对的）。转换器通常会在`L<>`链接周围增加一些词，例如“`L<foo(1)>`”会变成“`the foo(1) manpage`”。所以，如果你希望转换后的文档是可读的，不要写类似这样的“`the L<foo> manpage`”的文本，因为最后这会得到“`the the foo(1) manpage manpage`”。

如果需要完全控制链接中的文本，应当使用`L<show this text|foo>`形式。

标准`podchecker`程序会检查pod语法，生成错误和警告。例如，它会检查未知的pod序列，以及看上去像空行但实际上包含空白符的行。不过还是建议你吧文档传入两个或更多不同的pod转换器，检查结果是否正确。你发现的一些问题可能只是某些特定转换器特有的问题，对于这些问题，你可能想解决，也可能并不想解决。下面的pod存在一些问题：

```
=encoding utf8

=pod

This is a D<para>.

=item * This is an item

=cut
```

使用`podchecker`捕获到两个错误，另外给出一个警告，指示空行中存在空白符：

```
% podchecker broken.pod
*** ERROR: Unknown interior-sequence 'D' at line 5 in file broken.pod
*** ERROR: =item without previous =over at line 7 in file broken.pod
*** WARNING: line containing nothing but whitespace in paragraph at line 8
in file broken.pod
broken.pod has 2 pod syntax errors.
```

与以往一样，一切都可能由于普通黑客的随心所欲而改变。

# 为Perl程序建立文档

希望你能为代码建立文档，而不论你是否是一个普通黑客。如果这么做，你可能希望在pod中包含以下节：

`=head1 NAME`

程序或模块的名字。

`=head1 SYNOPSIS`

模块用法小结。

`=head1 DESCRIPTION`

文档的大块描述（在这个上下文中，称之为“大块”（Bulk）很合适）。

`=head1 AUTHOR`

你是谁（或者如果你对你的程序没有自信，可以提供一個别名）。

`=head1 BUGS`

你做错了什么（以及为什么实际上这不是你的错）。

`=head1 SEE ALSO`

人们在哪里能找到相关的信息（以便他们解决你的bug）。

`=head1 COPYRIGHT`

版权声明。如果你想声明一个明确的版权，可以写为：

Copyright 2013, Randy Waterhouse. All Rights Reserved.

很多模块还会增加以下文本：

This program is free software. You may copy or  
redistribute it under the same terms as Perl itself.

注意：如果打算把你的pod放在文件末尾，而且使用了一个`__END__`或`__DATA__` token，要确保在第一个pod指令前面加一个空行：

`__END__`

`=head1 NAME`

Modern - I am the very model of a modern major module

如果`=head1`前没有空行，pod转换器会忽略pod文档的开头，而不论你的pod文档多么丰富、多么准确、多么有文化内涵。



# Perl文化

这本书只是Perl文化的一部分，所以我们不打算把关于Perl文化所了解的一切都放在这里。我们只是通过一点点历史、一点点艺术（有人可能会说“一丁点艺术”），以及Perl社区的一些精粹来吊起你的胃口。要想更多地了解Perl文化，还请参见<http://www.perl.org>。或者也可以认识一些Perl程序员。我们无法告诉你他们是哪些人，不过Perl程序员都有一个共同的特点，这就是他们都特别热心，非常乐于助人。

## 历史决定成败

要了解Perl为什么会以这种方式定义（或者为什么不以其他方式定义），首先必须了解Perl存在的理由。所以让我们找出已经落满灰尘的历史书……

还要从1986年说起，Larry当时是一个系统程序员，正在参与一个开发多级安全广域网的项目。他负责系统安装，包括西海岸的3台VAXen和3台Sun，通过一个加密的1200波特串行线路连接到东海岸类似的配置上。由于Larry的主要任务是提供支持（他并不是这个项目的程序员，只是系统专家），所以他能充分利用他的3大“品质”（懒惰、急躁和傲慢）来开发和改进各种有用的工具，如`rn`、`patch`和`warp`<sup>注1</sup>。有一天，Larry刚刚把`rn`分解得支离破碎，总经理进来对他说“Larry，我们需要为这6个VAXen和6个Sun提供一个配置管理和控制系统。一个月就要，动手做一个吧！”

---

注1：大概就是这个时候Larry深刻体会到特性蔓延（feeping creaturism）的含义，由于总是克制不住地想要“再增加一个特性”，他绝望地想找出这种冲动是否有生理需求方面的原因。毕竟，既然生活这么复杂，程序是不是也可以很复杂呢？特别是像`rn`之类的程序，应该把它当作一个高级人工智能项目，从而能为你读新闻。当然，有些人会说`patch`程序已经太聪明了。

Larry是一个从不逃避工作的人，他问自己：要建立这样一个面向东西海岸的CM系统，如果不从头开始写代码，要想让东西海岸都能查看问题报告，而且都能完成核准和控制，最好的办法是什么。他找到了一个答案，只有一个词：B-news<sup>注2</sup>。Larry开始在这些机器上安装新闻软件，并增加了两个控制命令：一个是“append”命令，用来追加一个现有的文章，另外一个“synchronize”命令，保证东西海岸的文章数相同。CM可以使用RCS（版本控制系统，Revision Control System）来完成，核准和提交则使用新闻软件和`rn`完成。到目前为止一切都很好。

然后总经理又要求他生成报表。新闻都在一个主机上由多个单独的文件来维护，文件之间存在大量交叉引用。Larry的第一个想法是“可以用`awk`”。遗憾的是，那时的`awk`还不能根据文件中的信息处理多个文件的打开和关闭。Larry不想为此编写一个特殊用途的工具。因此，一种新的语言诞生了。

这个新工具原先的名字并不是Perl。Larry和他的同事和亲友一起考虑了很多名字（包括写这一章历史的Dan Faigin，还有他的妹夫Mark Biggar，他对最初的设计也有很大帮助）。Larry实际上考虑过字典中的所有3字母和4字母单词，不过都否决了。最初的名字是“Gloria”，这是他最亲爱的妻子的名字。很快他发现这会在家带来太多混乱。

后来名字变成了“Pearl”，再后来演变成现在的“Perl”，部分原因是Larry见过另一个语言的资料，那个语言就名为PEARL。不过主要是因为他太懒了，不想每次都输入5个字母。当然，现在Perl也可以用作为一个4字母的单词（不过，还是可以注意到原首字母缩写PEARL的痕迹：实用摘录与报表语言“Practical Extraction And Report Language”<sup>注3</sup>）。

最早的Perl缺少当今Perl的很多特性。那时已经有了模式匹配和文件句柄，标量和格式也有了，但是函数很少，还没有关联数组，而且正则表达式的实现还很成问题（借用自`rn`）。当时手册页只有15页。但Perl比`sed`和`awk`都快，并且开始在那个项目的其他应用中得到使用。

不过大家都需要Larry。有一天另一位经理过来对Larry说，“Larry，支持我们的研发吧。”Larry点头答应。他带去了他的Perl，发现它变成了一个很棒的系统管理工具。他借用了Henry Spencer漂亮的正则表达式包，并切割肢解（可能让Henry很倒胃口）。然后Larry增加了他想要的大部分特性，以及其他需要的一些特性。Larry把它发布到网上<sup>注4</sup>。正

---

注2： 也就是Usenet传输软件的第二个实现。

注3： 有时这称为一个反向缩略语（backronym），因为最初的名字又有扩展定义，使它有了新的含义。

注4： 更让人惊奇的是，尽管后来他又任职于Jet Propulsion Lab，然后加入NetLabs和Seagate，再然后又在O'Reilly & Associates工作（这是一家出版计算机及相关内容技术手册的小公司，现在名为O'Reilly Media），但他还在继续发布perl。



如他们所说，接下来的就是历史了<sup>注5</sup>。情况基本是这样的：Perl 1.0于1987年12月18日发布，有些人还很正式地把这一天当作Perl的生日。Perl 2.0随后于1988年6月发布，Randal Schwartz创造了传说中的“Just Another Perl Hacker”（JAPH）签名。1989年，Tom Christiansen在巴尔的摩的Usenix论坛会议上提供了第一本公开的Perl教程。随着Perl 3.0于1989年10月发布，这个语言首先在GNU公共许可条款下发布和发行。

1990年3月，Larry写了第一首Perl诗（参见下一节）。然后他和Randal写了本书的第一版“The Pink Camel”；这本书于1991年初出版<sup>注6</sup>。Perl 4.0同期发布；除了GPL，它还包含了一个Artistic许可。在Perl 4之后，Larry构想了一个改进的全新Perl；1994年，成立了Perl 5 Porter小组或简称为p5p，可以将perl移植到几乎任何一个系统上。这个小组（虽然人员在流动）仍在负责Perl的开发和技术支持。

万众期待的Perl 5终于在1994年10月问世。这是一个完全重写的Perl版本，包含了对象和模块。Perl 5的到来甚至在“The Economist”中都有提到<sup>注7</sup>。1995年，CPAN正式引入到Perl社区。Jon Orwant在1996年开始出版《The Perl Journal》杂志。经过长时间的酝酿，这本书的第二版“The Blue Camel”终于在那个秋天问世。1997年，一些知名的Perl高手创立了The Perl Institute来组织Perl的推广和支持活动。

第一届O'Reilly Perl Conference (TPC) 于1997年夏天在加利福尼亚的圣何塞召开。在这次大会上，一群纽约人成立了第一个Perl用户组，他们把它叫做/New York Perl M((o|u)ngers|aniacs)\*/, 不过这有点奇怪，所以后来变成了NY.pm，以后的大多数Perl用户组都照此模式起名。下一年这个组织帮助人们建立他们自己的用户组时，又改名为Perl mongers。Perl mongers接管了The Perl Institute。

1999年，Kevin Lenzo在匹兹堡的卡耐基梅隆组织召开了Yet Another Perl Conference (YAPC)。这些技术大会主要在美国西海岸靠近硅谷的地方召开。这对于东海岸的人来说不太方便。所以同年Chris Nandor写了一个Perl脚本为波士顿红袜队的游击手诺马·贾西亚帕拉提交了25000张全明星赛选票<sup>注8</sup>，这使得以后数年很多全明星阵容都有他的身影，一些人相信这也促使电视剧《体育之夜》增加了这样一个小情节<sup>注9</sup>。

---

注5： 这可以算是对历史的一个注解。开发Perl时，Larry刚刚拆分了`rn`，他本想对`rn`做一次全面重写。由于一直投身于Perl，Larry再没有时间去碰`rn`。所以到目前为止`rn`仍是那时支离破碎的状态。有时，Larry也威胁说要用Perl重写`rn`（不过不是当真的）。

注6： 书名叫《Programming perl》，“perl”全是小写。

注7： “不同于大多数其他免费可用的软件，Perl很有用，而且可以用。” “Electric metre”，*The Economist*, July 1, 1995。

注8： “网络植入仍对全明星投票构成威胁，” ESPN.com。

注9： 在“Louise Revisited”这一集中（于1999年10月26日播放），Jeremy使用一个Perl脚本为Casey（一位主播）投票。



第二年，伦敦Perl mongers组织了YAPC::EU（不过这不是欧洲的第一件Perl大事，要知道第一届德国Perl Workshop甚至比The Perl Conference还要早）。这些会议取得了巨大成功，后来发展成为美国的Yet Another Foundation（也称为The Perl Foundation）和欧洲的YAPC Europe Foundation。很快亚洲和南美也有了YAPC，不过实际上这些都是不同的会议，只是名字相同而已。现在很难找到哪一周没有Perl大事发生，这有利于建立一个紧密关联的Perl社区，尽管人们主要还是单独工作，不过会经常汇合在一起。

The Perl Conference后来进一步延伸到其他主题，变成了The Open Source Conference或简称为OSCON，Larry经常在这里发表他的“State of the Onion”演说，Damian Conway则利用“The Conway Channel”打动听众。在2000年的OSCON上，Larry宣布了Perl 6，这是一个从头开始的雄心勃勃的项目（这本书决定不介绍这个内容）。在这本书中，我们只是谈到Perl 6很有意思，它完全改造了Perl 5开发，与我们这里谈到的Perl完全不同，只不过同名而已。实际上它完全是另一种语言，窃取了Perl的特性，就像Perl从其他语言窃取特性一样。

要了解更多的Perl历史（至少2002年前的历史），可以查看CPAST上的Perl大事纪年表，CPAST全名是Comprehensive Perl Arcana Society Tapestry (<http://history.perl.org>)。

## Perl诗歌

Perl假设它遇到的任何裸字（bareword）最终都会成为一个已定义的子例程的名字，即使你还没有定义这个子例程。有时这称为“Perl诗歌模式”，允许人们用Perl写诗，比如下面这个怪东西：

```
BEFOREHAND: close door, each window & exit; wait until time.
    open spellbook, study, read (scan, select, tell us);
write it, print the hex while each watches,
    reverse its length, write again;
    kill spiders, pop them, chop, split, kill them.
        unlink arms, shift, wait & listen (listening, wait),
sort the flock (then, warn the "goats" & kill the "sheep");
    kill them, dump qualms, shift moralities,
    values aside, each one;
        die sheep! die to reverse the system
        you accept (reject, respect);
next step,
    kill the next sacrifice, each sacrifice,
    wait, redo ritual until "all the spirits are pleased";
    do it ("as they say").
do it(*everyone***must***participate***in***forbidden**s*e*x*).
return last victim; package body;
    exit crypt (time, times & "half a time") & close it,
    select (quickly) & warn your next victim;
AFTERWORDS: tell nobody.
    wait, wait until time;
```

```
wait until next year, next decade;
    sleep, sleep, die yourself,
    die at last
```

Larry写了这首诗，并把它发送给*news.groups*，希望支持他的请求，创建一个*comp.lang.perl.poems*组。很多人可能注意到那是4月1日（愚人节），不过这并不妨碍人们写更多的Perl诗歌。

Sharon Hopkins写了很多Perl诗歌，另外关于Perl诗歌还写了一篇文章在1992年冬的Usenix技术大会上发表，名为“Camels and Needles: Computer Poetry Meets the Perl Programming Language”。除了作为最多产的Perl诗人，Sharon发表的作品也最多，下面这首诗在《The Economist》和《The Guardian》上都刊登过：

```
#!/usr/bin/perl

APPEAL:

listen (please, please);

open yourself, wide;
    join (you, me),
connect (us,together),

tell me.
do something if distressed;

@dawn, dance;
@evening, sing;
read (books,$poems,stories) until peaceful;
study if able;

write me if-you-please;

sort your feelings, reset goals, seek (friends, family, anyone);

do*not*die (like this)
if sin abounds;

keys (hidden), open (locks, doors), tell secrets;
do not, I-beg-you, close them, yet.

        accept (yourself, changes),
        bind (grief, despair);

require truth, goodness if-you-will, each moment;

select (always), length(of-days)

# listen (a perl poem)
# Sharon Hopkins
# rev. June 19, 1995
```

# Perl程序员的品质

## 懒惰

懒惰 (Laziness) 听上去像是一个恶习，不过这里的懒惰有所不同。通常的懒惰是总想逃避手上的工作，而懒惰品质则是避免将来的工作。对于同样的任务，熟练使用Perl的程序员会创建一些工具，从而能更容易地完成这些任务。Perl是一个自动完成任务的强大语言，当前自动完成的工作越多，程序员将来手动完成的工作就越少。

## 急躁

急躁 (Impatience) 是指，如果计算机做了它想做的事而不是你想做的事，你就会有这种不愉快的感觉。嗯，更准确地讲，如果软件另一端的程序员选择了错误的默认设置，建立了一个糟糕的GUI，或者没有提供访问数据的途径，你就会充分感受到这种痛苦，而且不希望其他程序员也遭受相同的痛苦，不要再为你浪费的时间而纠结，应该充分利用这些时间做一些对其他人有益的事情。

## 傲慢

傲慢 (Hubris) 是指，利用正确的工具你可以做任何事情。这只不过是个简单的编程问题，不是吗？傲慢也可能让你飞得太高（甚至接近太阳）而忘乎所以。

# 大事记

一年中几乎每一周都有某个Perl大事发生。下面是一些主要的大事，大多都会在The Perl Review社区日历 ([http://theperlreview.com/community\\_calendar](http://theperlreview.com/community_calendar)) 上列出。

## *The Perl Conference, OSCON*

O'Reilly & Associates在1997年召开的The Perl Conference并不是第一件Perl大事，不过从历史看来，这可能是最重要的一件大事。在这次会议上，一群纽约人建立了第一个Perl用户组NY.pm。紧接着又有另外一些Perl mongers用户组随之创建。短短几年间，已经有了数百个用户组。The Perl Conference进一步扩展成为The Open Source Conference，或简称为OSCON。

## YAPC

YAPC或Yet Another Perl Conference有多种形式，至少在4大洲都有活动。每年都会在亚洲、欧洲、北美洲和南美洲召开这些低成本的草根型会议（大多是非商业性的）。尽管它们的名字一样，但分别由不同的小组组织。

## *Perl研讨会 (Perl Workshop)*

尽管YAPC为时数天，不过Perl研讨会通常只有一到两天，主要讨论某个特定的主题，如Perl QA Workshop的重点是CPAN基础设施和Perl测试的问题。很多人可能不



知道German Perl Workshop才是第一个有组织的Perl大事，它甚至比Perl mongers或Perl Conference还要早。

### 专家活动 (Hackathons)

所有Perl大事中最没有组织的是专家活动 (hackathons)，Perl人员汇聚到同一个地方工作。有时这些专家活动会强调某个特定的主题，有时则只是一群人在同一个房间里分别完成他们自己的项目。

## 获得帮助

Perl人员是最热心的人，甚至不喜欢Perl的人也开始认识到这一点。我们认为，Perl来源于如此之多不同的语言，这一点吸引了那些喜欢不同种类语言（而不只是他们知道的那种语言）的人。可能他们发现每个语言都有一点好处。

如果需要帮助，会有很多人在等着伸出援手，互联网上关于Perl的各种讨论形式几乎应有尽有。下面给出几个比较著名的网站：

<http://perldoc.perl.org>

所有Perl文档都是在线的，所以不用担心没有Perl文档（尽管你的平台和封装系统可能不这么想）。没错，有些开发商提供perl时就不提供手册。

### Learn Perl

可以将这个网站作为你的起点，初学者从中可以得到很多可用的资源，包括我们这里所列的一些资源。

### Perl初学者邮件列表 (Perl beginners mailing list)

Casey West启动了 this 邮件列表，使初学者可以在一个安全的环境中问最基本的问题。其他论坛的管理可能稍有欠缺，可能会让新Perl程序员很受打击。

### Perlmonks

Perlmonks是一个专门针对Perl的web公告板。它并不是一个问讯台，不过如果你做好了准备工作，问了一个好问题，可能很快会得到顶尖高手的帮助。你可能想先读一读“brian's Guide to Solving Any Perl Problem”<sup>注10</sup>。

### Stackoverflow

Stackoverflow是一个面向一般编程的问答网站。尽管不是专攻Perl，不过这里有很多Perl专家，经常来网站回答问题。

### 当地的Perl mongers用户组

全世界有数百个Perl mongers用户组。他们各有风格，可以寻找离你近（或不太近）的Perl用户，并与他们交流。这些用户组大多会组织研讨会和其他会议。可以在

---

注10：《Mastering Perl》中也给出了这个指南。

<http://www.pm.org>找到离你最近的Perl mongers，如果找不到，完全可以自己启动一个！

### Usenet新闻组

Perl新闻组是一个很棒的Perl信息来源（有时也有些混乱）。你的第一站可以是 *news:comp.lang.perl.moderated*，这是一个规模适中的低流量新闻组，包含通知和技术讨论。由于规模适中，这个新闻组很适合阅读。

高流量的 *news:comp.lang.perl.misc* 组会讨论各种问题，从技术问题到Perl哲学，再到Perl游戏，以至Perl诗歌都有讨论。就像Perl本身一样，*news:comp.lang.perl.misc* 的关键是必须有用，不会问太傻的问题<sup>注11</sup>。

如果你使用一个Web浏览器访问Usenet，而不是一个常规的新闻阅读器，需要在新闻组名前面加上 *news:* 来访问指定的某个新闻组（前提是要有一个新闻服务器）。或者，如果你使用一个Usenet搜索服务，如Google Groups，需要指定 *\*perl\** 作为搜索的新闻组。

### 邮件列表

很多主题（包括一般主题和特定主题）都有专门的邮件列表，大多会在 <http://lists.perl.org> 上列出。也可以直接从项目网站找到其他一些邮件列表。还可以使用诸如 <http://markmail.org> 等网站在多个Perl列表中搜索归档。

### IRC

Internet Relay Chat (IRC) 也是Perl程序员喜欢的一种交流方式。如果你喜欢这种方式，会发现可以与很多人交流。这些聊天室并不把自己当作问讯台，所以如果直接进去问个问题而不自我介绍，这有点不礼貌，就像贸然闯入一个聚会。不过，还有一些专门的帮助渠道，如 *#perl-help* 和 *#win32*。可以在 <http://www.irc.perl.org/> 找到更多IRC渠道。

---

注11：当然，有些问题太傻不值得回答（特别是已经在线手册页和FAQ中回答过的问题。与输入问题相比，既然可以用更短的时间找到同样的答案，为什么还要在新闻组上问这个问题呢）。

# 参考资料





# 特殊名

这一章介绍对Perl有特殊含义的变量。大多数标点符号名在shell中都有某种合理的助记符或对应的符号（或者这二者都有）。不过，如果你想使用比较长的变量名作为同义词，可以在程序最前面做以下声明：

```
use English "-no_match_vars";
```

这会为所有短名设置别名，将其别名指定为当前包中的长名。其中一些变量甚至还有中间名，这通常是从`awk`借用的。大多数人最后还是会使用短名，至少当前常用的变量会使用短名。在这本书中，我们一直使用的都是短名，不过也经常提到长名（放在括号里），以便你在这一章中可以很容易地查找这些名字。

这些变量的语义可能有魔法（要创建你自己的魔法，可以参见第14章）。这些有特殊含义的变量很多都是只读的。如果你想对那些变量赋值，会产生一个异常。

下面我们首先提供一个简明的变量和函数列表，Perl为这些变量和函数赋予了特殊的含义，这里将它们按类型分组，即使你不确定哪一个名字合适，也可以由此查找变量。然后再按适当名字（或者相对来讲最合适的名字）的字母顺序解释所有这些变量。

## 按类型分组的特殊名

“类型”这个词的使用很不严格，更确切地讲，这里是按作用域对变量分组，也就是变量的可见范围。

### 正则表达式特殊变量

下面的特殊变量与模式匹配相关，在模式匹配的动态作用域中可见。换句话说，它们就好

像是用`local`声明的一样，所以你不需要用`local`来声明这些变量。参见第5章。

```
$digits

$& ($MATCH)
$' ($POSTMATCH)
`${ ($PREMATCH)

${^MATCH}
${^POSTMATCH}
${^PREMATCH}

$+ ($LAST_PAREN_MATCH)
%+ (%LAST_PAREN_MATCH)
@+ (@LAST_MATCH_END)

@-
%-

$^R ($LAST_REGEXP_CODE_RESULT)
$^N ($LAST_SUBMATCH_RESULT)
```

## 各个文件句柄的变量

这些特殊变量从不需要用`local`声明，因为它们指示的值总是属于当前选择的输出文件句柄，每个文件句柄都有自己的一组值。用`select`选择另一个文件句柄时，原文件句柄会记住这些变量原来的值，变量现在反映的是新文件句柄的值。参见`IO::Handle`模块。

```
$| ($AUTOFLUSH, $OUTPUT_AUTOFLUSH)
$- ($FORMAT_LINES_LEFT)
$= ($FORMAT_LINES_PER_PAGE)
$~ ($FORMAT_NAME)
$% ($FORMAT_PAGE_NUMBER)
$^ ($FORMAT_TOP_NAME)
```

## 各个包的特殊变量

这些特殊变量在各个包中独立存在。没有必要使用`local`声明这些变量，因为`sort`会自动对`$a`和`$b`局部化，其余的变量最好不要去干涉（不过如果使用了`use strict`，可能需要用`our`来声明这些变量）。

```
$a
$AUTOLOAD
$b
@EXPORT
@EXPORT_OK
%EXPORT_TAGS
%FIELDS
@ISA
%OVERLOAD
$VERSION
```



## 程序范围特殊变量

从最完全意义上讲，这些变量才是真正的全局变量。在各个包中它们都表示同一个东西，因为只要这些变量未限定，它们都会强制放在包main中（但@F除外，它在main中很特殊，但不强制如此）。如果想得到其中某个变量的一个临时副本，必须在当前动态作用域中对该变量局部化。

```
%ENV
%! (%ERRNO, %OS_ERROR)
%INC
%SIG
%^H

@_
@ARGV
@INC

$_
$0 ($PROGRAM_NAME)
$ARGV

$! ($ERRNO, $OS_ERROR)
$" ($LIST_SEPARATOR)
$$ ($PID, $PROCESS_ID)
$( ($GID, $REAL_GROUP_ID)
$) ($EGID, $EFFECTIVE_GROUP_ID)
$, ($OFS, $OUTPUT_FIELD_SEPARATOR)
$. ($NR, $INPUT_LINE_NUMBER)
$/ ($RS, $INPUT_RECORD_SEPARATOR)
$: ($FORMAT_LINE_BREAK_CHARACTERS)
$; ($SUBSEP, $SUBSCRIPT_SEPARATOR)
$< ($UID, $REAL_USER_ID)
$> ($EUID, $EFFECTIVE_USER_ID)
$? ($CHILD_ERROR)
$@ ($EVAL_ERROR)
$[
$\ ($ORS, $OUTPUT_RECORD_SEPARATOR)
$]
$^A ($ACCUMULATOR)
$^C ($COMPILING)
$^D ($DEBUGGING)
${^ENCODING}
$^E ($EXTENDED_OS_ERROR)
${^GLOBAL_PHASE}
$^F ($SYSTEM_FD_MAX)
$^H
$^I ($INPLACE_EDIT)
$^L ($FORMAT_FORMFEED)
$^M
$^O ($OSNAME)
${^OPEN}
$^P ($PERLDB)
$^R ($LAST_REGEXP_CODE_RESULT)
${^RE_DEBUG_FLAGS}
```

```

${^RE_TRIE_MAXBUF}
$^S (EXCEPTIONS_BEING_CAUGHT)
$^T ($BASETIME)
${^TAINT}
${^UNICODE}
${^UTF8CACHE}
${^UTF8LOCALE}
$^V ($PERL_VERSION)
$^W ($WARNING)
${^WARNING_BITS}
${^WIDE_SYSTEM_CALLS}
${^WIN32_SLOPPY_STAT}
$^X ($EXECUTABLE_NAME)

```

## 各个包的特殊文件句柄

除了DATA以外（每个包都有一个DATA），如果以下文件句柄没有用其他包名完全限定，则假设它们总在main中：

```

_ # (underline)
ARGV
ARGVOUT
DATA
STDIN
STDOUT
STDERR

```

## 各个包的特殊函数

下面的子例程名对Perl有特殊的含义。它们会由于某个事件隐式调用，如访问一个绑定变量或尝试调用一个未定义的函数。这一章我们不做介绍，因为这个内容在本书其他地方已经做过详细说明。

未定义的函数调用拦截器（参见第10章）：

```
AUTOLOAD
```

垂死对象的最终化（参见第12章）：

```
DESTROY
```

异常对象（参见第27章中的die）：

```
PROPAGATE
```

自动初始化和自动清理函数（参见第16章）：

```
BEGIN, CHECK, UNITCHECK, INIT, END
```

线程支持：

CLONE, CLONE\_SKIP

绑定方法（参见第14章）：

BINMODE, CLEAR, CLOSE, DELETE, DESTROY, EOF, EXISTS, EXTEND, FETCH, FETCHSIZE, FILENO, FIRSTKEY, GETC, NEXTKEY, OPEN, POP, PRINT, PRINTF, PUSH, READ, READLINE, SCALAR, SEEK, SHIFT, SPLICE, STORE, STORESIZE, TELL, TIEARRAY, TIEHANDLE, TIEHASH, TIESCALAR, UNSHIFT和WRITE.

## 按字母顺序排列的特殊变量

我们根据长变量名按字母顺序排列这些变量。如果你不知道一个变量的长名，可以从上一节中找到。（没有字母名字的变量排在最前面）。

所以下面不用重复，每个变量描述都以某一个或多个标注开头（见表25-1）。

表25-1：特殊变量的标注

标注	含义
XXX	过期，新代码中不再使用
NOT	不正式使用（只做内部使用）
RMV	已从Perl删除
ALL	真正的全局，由所有包共享
PKG	包全局。每个包可以有它自己的一组变量
FHA	文件句柄属性。每个I/O对象有这样一个属性
DYN	自动为动态作用域（隐含ALL）
LEX	编译时为词法作用域
RO	只读。如果修改这个变量会产生一个异常

列出多个变量名或符号时，默认地只能使用短名。如果使用了English模块，则较长的同义词变量在当前包中也可用，不过只对当前包可用，即使变量标为[ALL]。

形如`method HANDLE EXPR`的项为IO::Handle模块提供的各文件句柄的变量显示面向对象接口。在v5.14中，这个模块会根据需要加载（如果你愿意，还可以使用`HANDLE->method(EXPR)`记法）。这样一来，检查或修改变量之前就不必调用select来修改默认的输出句柄。这些方法会返回属性原来的值。如果提供了EXPR参数，则设置一个新值。如果没有提供EXPR参数，大多数方法对当前值不做任何处理，但autoflush除外，它会假设参数为1（这么做可能只是想与众不同）。

`$_` [ALL] 默认输入和模式搜索空间。下面的每对代码是等价的：



```

while (<>) {...} # 仅在无修饰while测试中等价
while (defined($_ = <>)) {...}

chomp
chomp($_)

/^Subject:/
$_ =~ /^Subject:/

tr/a-z/A-Z/
$_ =~ tr/a-z/A-Z/

```

以下函数如果没有指定操作对象，Perl就会假设操作对象为\$\_：

- 列表函数（如print和unlink）、一元函数（如ord、pos和int），以及所有文件测试，但-t除外，它的操作对象默认为STDIN。所有默认处理\$\_的函数会在第27章中相应标注。
- 模式匹配操作m//和s///，以及转换操作y///和tr///（如果未同时用=~操作符）。
- foreach循环（甚至写作for循环或者用作为一个语句修饰符）的迭代器变量（如果没有提供其他变量）。
- grep和map函数中的隐式迭代器变量（没有办法为这些函数指定其他变量）。
- <FH>、readline或glob操作测试自己的结果作为while测试的唯一标准时，放置输入记录的默认位置会假设为\$\_。这个赋值不会出现在while测试之外，或者如果while表达式中包含任何其他元素，那么也不会出现这个赋值。

由于\$\_是一个全局变量，有时可能会导致不受欢迎的副作用。在v5.10中，可以用my声明使用\$\_的一个私有（词法作用域）版本。另外，声明our \$\_会在当前作用域中恢复全局\$\_。

助记方法：下划线是某些操作的底层操作数。

@\_ [ALL] 在一个子例程中，这个数组包含传入该子例程的参数表。参见第7章。

\_ (下划线)

[ALL] 这是一个特殊的文件句柄，用来缓存来自最后一个成功的stat、lstat或文件测试操作符（如-w \$file或-d \$file）的信息。

\$digits

[DYN, RO] 编号变量\$1、\$2等（最大可以达到你想要的任意大小）<sup>注1</sup>，其中包含当前活动动态作用域的最后一个匹配模式中与小括号匹配对应的文本。

助记方法：如\digits。

\$] [ALL] 返回版本（version）+ 补丁级别（patchlevel）/1000。可以在脚本开始位置使

注1： 尽管很多正则表达式引擎只支持最多9个反向引用，但Perl没有这个限制。所以，如果你写为\$768，Perl也不会介意，不过你的代码维护者可能会很不满。

用，来确定执行这个脚本的Perl解释器是否在正确的版本范围内（助记方法：Perl的这个版本在中括号以内吗）。示例：

```
warn "No checksumming!\n" if $] < 3.019;  
die "Must have prototyping available\n" if $] < 5.003;
```

参见`use VERSION`和`require VERSION`的文档，其中给出了一种便利的方法，可以在Perl解释器版本太老时妥善地退出。参见`$^V`，它提供了一个更灵活的Perl版本表示。

**\$[** [XXX, LEX] 数组中第一个元素以及子串中第一个字符的索引。默认为0，不过我们经常将它设置为1，使得Perl在建立下标以及计算`index`和`substr`函数时表现得更像`awk`（或`FORTRAN`）。因为直接对`$[`赋值太危险，所以现在把它处理为一个词法作用域编译器指令，它不能影响任何其他文件的行为（助记方法：[开始下标]）。

**\$#** [RMV, ALL] 已经在v5.10版本中删除。不要使用这个变量，而应当使用`printf`。`$#`原先包含所打印数字的输出格式，试图模拟`awk`的`OFMT`变量（助记方法：`#`是数字记号，不过如果你聪明，就该把它忘掉，以免受到打击）。

这不是放在数组名前面用来得到最后一个索引的印记，如`$#ARRAY`。在Perl中仍然可以用`$#ARRAY`得到数组的最后一个索引。这二者相互之间没有任何关系。

**\$\*** [RMV, ALL] 这个变量已经删除，原先可以将它设置为`true`，使Perl假设所有没有显式`/s`的模式匹配上都有`/m`。在v5.10版本中已经将它删除（助记方法：`*`多次匹配）。

**%-** [DYN, RO] 类似于`%+`（`%LAST_PAREN_MATCH`），这个变量允许访问当前活动动态作用域的最后一个成功模式匹配中的命名捕获组。键为捕获组的名，值为数组引用。分别包含所有同名组匹配的值（如果有多个），其顺序与这些名字在模式中出现顺序一致。

对这个散列使用`each`调用的同时不要在循环本身完成模式匹配，否则结果会出乎你的意料。

如果你不想写为类似`$_{-{NAME}}[0]`的形式，可以使用标准`Tie::Hash::NamedCapture`模块为`%-`提供你自己选择的一个别名。

**\$a** [PKG] `sort`函数使用这个变量来保存要比较的两个值中的第一个值（第二个值是`$b`）。`sort`操作符会编译到`$a`所在的同一个包，但`sort`操作符的比较函数不一定也编译到这个包中。这个变量会在`sort`比较模块中隐式地局部化。因为它是全局变量，所以即使有`use strict`也不会带来警告。因为这是具体数组值的一个别名，你可能认为可以修改这个变量，不过请不要修改。参见`sort`函数。

## **\$ACCUMULATOR**

**^A** [ALL] `format`行的`write`累加器的当前值。格式包含`formline`命令，`formline`命令将其结果放在`^A`中。调用其格式后，`write`会打印`^A`的内容，并将它清空。所以除非你自己调用`formline`然后查看`^A`，否则你不会看到`^A`的内容。参见`formline`函数。



## ARGV

[ALL] 这是一个特殊的文件句柄，会迭代处理@ARGV中的命令行文件名。通常写为放在尖角操作符中的null文件句柄：<>。

## \$ARGV

[ALL] 使用<>或readline操作符读ARGV句柄时，这个变量包含当前文件名。

## @ARGV

[ALL] 这个数组中包含提供给脚本的命令行参数。注意 \$#ARGV 通常是参数个数减1，因为第一个参数是 \$ARGV[0]，而不是命令名。可以使用 scalar@ARGV 来得到程序参数的个数。程序名可以用 \$0 得到。

## ARGVOUT

[ALL] 在 -i 开关或 \$^I 变量下处理 ARGV 句柄时，会使用这个特殊的文件句柄。参见第17章的 -i 开关。

## \$AUTOLOAD

[PKG] 这是各个包的变量，执行 AUTOLOAD 方法期间，这个变量包含运行 AUTOLOAD 方法所代表的函数的完全限定名。参见第25章。

\$b [PKG] 这个变量与 \$a 搭配，在 sort 比较中使用。详细内容参见 \$a 和 sort 函数。

## \$BASETIME

\$^T [ALL] 脚本开始运行的时间，单位为秒，要从纪元时间开始计算（对于 Unix 系统，这就是1970年1月1日0时）。-M、-A 和 -C 文件测试返回的值都是相对于这个时刻的时间。

## \$CHILD\_ERROR

\$? [ALL] 最后一个管道 close、反引号 (``) 命令或者 wait、waitpid 或 system 函数返回的状态。注意，这不只是简单的退出码，而是由底层 wait(2) 或 waitpid(2) 系统调用（或其等价调用）返回的一个完整的16位状态字。因此，子过程的退出值放在高字节上——即 \$? >> 8。在低字节上， \$? & 127 指出这个进程是因哪个信号（如果有）而结束，另外 \$? & 128 会报告进程终止是否会生成一个核心转储（助记方法：类似于 sh 及后代 shell 中的 \$?）。

在一个 END 块中， \$? 包含为 exit 提供的值。可以修改 END 中的 \$? 来改变脚本的退出状态。例如：

```
END {  
    $? = 1 if $? == 255; # 如果结束会使它变成255  
}
```

在 VMS 下，如果有 pragma use vmsish "status"， \$? 将反映真正的 VMS 退出状态，而不是默认模拟的 POSIX 状态。



如果C中支持h\_errno变量，某个gethost\*()函数失败时，h\_errno变量的数字值将通过\$?返回。

#### \$COMPILING

**\$^C** [ALL] 与-c开关关联的内部标志的当前值，主要结合-MO=...使用，使代码改变其行为。例如，你可能希望编译时不是使用正常的延迟加载，而是使用AUTOLoad自动加载，这样代码可以立即生成。设置**\$^C = 1** 类似于调用B::minus\_c。参见第16章。

#### DATA

[PKG] 这个特殊的文件句柄指示当前文件中\_\_END\_\_ token或\_\_DATA\_\_ token后面的任何内容。\_\_END\_\_ token总会打开main::DATA文件句柄，所以用在主程序中。\_\_DATA\_\_ token打开当前有效的包中的DATA句柄，所以不同的模块可能分别有自己的DATA文件句柄，因为它们（很可能）有不同的包名。

#### \$DEBUGGING

**\$^D** [ALL] 内部调试标志的当前值，在命令行上由-D开关设置；参见第17章中“开关”一节来了解可以有哪些值。与相应的命令行开关类似，可以使用数字或符号值，例如**\$^D = 10**或**\$^D = "st"**（助记方法：-D开关的值）。

#### \${^ENCODING}

[XXX, ALL] 用来将源代码转换为Unicode的Encode对象的对象引用。利用这个变量，不要求Perl脚本非得使用UTF-8编写。默认为undef。强烈建议不要直接处理这个变量。

这个变量是v5.8.2新增的。

#### \$EFFECTIVE\_GROUP\_ID

**\$)** [ALL] 这个进程的有效GID（组ID）。如果你使用的机器支持同时属于多个组，**\$)**可以提供你所在的组列表，各个组之间用空格分隔。第一个数由getegid(2)返回，后面则是getgroups(2)返回的数，其中某个数可能与第一个数相同。

类似地，赋给**\$)**的值也必须是一个用空格分隔的数字列表。第一个数用来设置有效GID，其余的（如果有）传入setgroups(2)系统调用。要想为setgroups提供空列表，为达到这个效果，只需要重复新的有效GID。例如，要求有效GID为5，另外要提供实际上为空的setgroups列表，可以写为：

```
$) = "5 5";
```

助记方法：小括号用来分组。如果你在运行setgid，有效GID就是你的组。需要说明：只有支持相应系统set-id例程的机器上才能设置**\$<**、**\$>**、**\$(和\$)**。**\$(和\$)**只能在支持setregid(2)的机器上交换。

#### \$EFFECTIVE\_USER\_ID

**\$>** [ALL] geteuid(2)系统调用返回的进程的有效UID。

举例：

```
$< = $>;          # 将真实UID设置为有效UID
($<,$>) = ($>,$<); # 交换真实UID和有效UID
```

可以使用POSIX::setuid同时改变有效UID和真实UID。要改为\$>，要求检查\$!以检测做出修改后可能的错误。

助记方法：如果你在运行setuid，这将成为你的UID。需要说明：\$<和\$>只能在支持setreuid(2)的机器上交换。有时甚至即使支持setreuid(2)也不能交换。

**%ENV [ALL]** 这个散列包含当前的环境变量。设置%ENV中的一个值会改变进程以及这个赋值之后启动的子进程的环境。（在类Unix的系统上，不能改变一个父进程的环境）。

```
$ENV{PATH} = "/bin:/usr/bin";
$ENV{PAGER} = "less";
$ENV{LESS} = "MQeicsnf"; # 我们最喜欢的less(1)开关
system "man perl";      # 选择新设置
```

要从你的环境删除某个变量，一定要使用delete函数，而不是将散列值设置为undef。

需要说明，作为crontab(5)记录运行的进程会继承一组缩减的环境变量。（如果你的程序从命令行运行得很好，但在cron下不能很好地运行，可能就是因为这个原因）。另外要注意，如果作为一个setuid脚本运行，还要设置\$ENV{PATH}、\$ENV{SHELL}、\$ENV{BASH\_ENV}和\$ENV{IFS}。参见第20章。

## **\$EVAL\_ERROR**

**\$@ [ALL]** 当前产生的异常，或者从最后一个eval操作得到的Perl语句错误消息（助记方法：语句错误“在(at)”哪里）。\$! (\$OS\_ERROR)会在失败时设置，但成功时并不清除，与它不同，\$@保证当最后一个eval有编译错误或运行时异常时会设置（为一个true值），倘若没有出现这些问题，\$@则保证会清除（为一个false值）。

警告消息不会收集到这个变量中。不过，你可以建立一个例程，通过设置\$SIG{\_\_WARN\_\_}来处理警告，如本节后面所述。

需要说明，\$@的值可能是一个异常对象而不是一个字符串。如果是这样，倘若这个异常对象类已经定义了字符串化重载，仍可以把它当作一个字符串。如果传播了一个异常，如下：

```
die if $@;
```

异常对象会调用\$@->PROPAGATE来看要做什么（字符串异常只会向字符串增加一个“propagated at”行）。

## **\$EXCEPTIONS\_BEING\_CAUGHT**

**\$^S [ALL]** 这个变量反映了解释器的当前状态，如果在一个eval中，返回true，否则返回



false。如果当前编译单元的解析还没有完成，则未定义，\$SIG{\_\_DIE\_\_}和\$SIG{\_\_WARN\_\_}处理器就可能是这种情况（助记方法：eval的状态）。

#### \$EXECUTABLE\_NAME

**\$^X** [ALL] 执行的Perl二进制版本的名字，来自C的argv[0]。

#### @EXPORT

[PKG] Exporter模块的import方法会查看这个数组变量，找到用use加载这个模块时或者使用:DEFAULT导入标记时默认要导出的其他包变量和子例程列表。如果声明了use strict，使用这个数组变量会发出警告，所以如果启用了这个pragma，必须使用our声明，或者用包名完全限定变量。不过，名字以字符串“EXPORT”开头的所有变量即使只使用一次也不会导致这个警告。参见第11章。

#### @EXPORT\_OK

[PKG] Exporter模块的import方法会查看这个数组变量，确定所请求的导入是否合法。如果声明了use strict，使用这个数组变量会发出警告。参见第11章。

#### %EXPORT\_TAGS

[PKG] 请求一个前面有冒号的导入符号时，如use POSIX ":sys\_wait\_h"，Exporter模块的import方法会查看这个散列变量。键是冒号标记，但不包括前面的冒号。值应当是一些数组引用，其中包含请求这个冒号标记时要导入的符号，这些符号都必须出现在@EXPORT或@EXPORT\_OK中。如果声明了use strict，使用这个变量会发出警告。参见第11章。

#### \$EXTENDED\_OS\_ERROR

**\$^E** [ALL] 当前操作系统特定的错误信息。在Unix下，**\$^E**等同于**\$!** (\$OS\_ERROR)，不过在OS/2、VMS和Microsoft系统以及MacPerl上，**\$^E**与**\$!** (\$OS\_ERROR)是不同的。具体内容参见你的移植版本信息。介绍**\$!**时提到的注意事项通常也适用于**\$^E**（助记方法：额外的错误解释）。

**@F** [PKG] 给定-a命令行开关时，输入行的字段会分解到这个数组。如果没有使用-a选项，这个数组没有特殊的含义（这个数组实际上只是@main::F，不是所有包中同时都有这个数组）。

#### %FIELDS

[XXX, PKG] 这个散列由fields pragma在内部使用，用来确定一个对象散列中的当前的合法字段。

#### format\_formfeed HANDLE EXPR

#### \$FORMAT\_FORMFEED

**\$^L** [ALL] write函数隐式输出这个变量，在表格首部前面完成一个换页。默认为"\f"。



`format_lines_left HANDLE EXPR`

[FHA] 当前选择的输出句柄的页面上剩余的行数，用于format声明和write函数（助记方法：`lines_on_page - lines_printed`）。

`format_lines_per_page HANDLE EXPR`

`$FORMAT_LINES_PER_PAGE`

`$=` [FHA] 当前选择的输出句柄的当前页长度（可打印的行），用于format和write。默认为60（助记方法：`=`有水平线）。

`format_line_break_characters HANDLE EXPR`

`$FORMAT_LINE_BREAK_CHARACTERS`

`$:` [ALL] 当前的一个字符集，字符串可以根据这个字符集分解，来填充格式中的连续字段（以`^`开始）。默认为"`\n-`"，这表示按空白符或连字符分解（助记方法：冒号是一个技术词汇，表示诗歌中一行的一部分。现在只要记住这个助记方法）。

`format_name HANDLE EXPR`

`$FORMAT_NAME`

`$~` [FHA] 当前选择的输出句柄的当前报表格式名。默认为文件句柄的名字（助记方法：在`$^`后面转弯）。

`format_page_number HANDLE EXPR`

`$FORMAT_PAGE_NUMBER`

`%%` [FHA] 当前选择的输出句柄的当前页号，用于format和write（助记方法：`%`是`troff(1)`中的页号寄存器。什么？你不知道`troff`是什么）。

`format_top_name HANDLE EXPR`

`$FORMAT_TOP_NAME`

`$^` [FHA] 当前选择的输出句柄的当前页首格式名。默认为文件句柄名再加上`_TOP`（助记方法：指向页首）。

`$^H` [NOT, LEX] 这个变量包含Perl解析器的词法作用域状态位（即提示）。这个变量只在内部使用。它的可用性、行为和内容很可能改变而不做任何通知。如果使用这个变量，你肯定会死得很惨，而且死因不明（助记方法：我们不给你提示）。

`%^H` [NOT, LEX] `%^H`散列提供与`$^H`相同的词法作用域语义，可以用于词法作用域pragma的实现。请仔细研读`$^H`下所列的可怕警告，另外还要再加一点：这个变量还是试验性的。

`%INC`

[ALL] 这个散列包含的元素对应于通过`do FILE`、`require`或`use`加载的各个Perl文件的文件名。键是你指定的文件名，值是实际找到的文件的位置。`require`操作符使用这个数组来确定一个给定的文件是否已经加载。例如：

```
% Perl-MLWP::Simple -le 'print $INC{"LWP/Simple.pm"}'  
/opt/perl/5.6.0/lib/site_perl/LWP/Simple.pm
```

## @INC

[ALL] 这个数组包含一个目录列表，`do FILE`、`require`或`use`可以在这些目录中找到Perl模块。最初它由-I命令行开关的参数和PERL5LIB环境变量中的目录组成，后面还包括默认Perl库，如下所示：

```
/usr/local/lib/perl5/site_perl/5.14.2/darwin-2level  
/usr/local/lib/perl5/site_perl/5.14.2  
/usr/local/lib/perl5/5.14.2/darwin-2level  
/usr/local/lib/perl5/5.14.2  
/usr/local/lib/perl5/site_perl  
.
```

后面的“.”表示当前目录。如果需要在程序中修改这个列表，可以使用`lib pragma`，这不仅能够在编译时修改变量，还可以加入依赖体系结构的相关目录（如包含XS模块所用共享库的目录）：

```
use lib "/mypath/libdir/";  
use SomeMod;
```

## \$INPLACE\_EDIT

**\$^I** [ALL] 原地编辑（inplace-edit）扩展的当前值。使用`undef`会禁用原地编辑。可以在程序中使用这个变量得到类似-I开关提供的行为。例如，要完成与这个命令等价的行为：

```
% Perl-i.orig -pe 's/foo/bar/g' *.c
```

可以在程序中使用以下等价代码：

```
local $^I = ".orig";  
local @ARGV = glob("*.c");  
while (<>) {  
    s/foo/bar/g;  
    print;  
}
```

助记方法：-i开关的值。

## \$INPUT\_LINE\_NUMBER

**\$.** [ALL] 读取（或调用了`seek`或`tell`）的最后一个文件句柄的当前记录号（通常是行号）。这个值可能不同于文件中的实际物理行号，这取决于“line”的实际概念——参见`$/`（`$INPUT_RECORD_SEPARATOR`）来了解如何影响这一点。对文件句柄显式调用`close`关闭文件会重置行号。因为`<>`从来都不会显式关闭，所以行号会跨`ARGV`文件增加（不过参见`eof`下的例子）。局部化`$.`也会使Perl的“最后一个读文件句柄”成为一个局部概念（助记方法：很多程序使用“.”表示当前行号）。

## \$INPUT\_RECORD\_SEPARATOR

`$/` [ALL] 输入记录分隔符，默认为换行符，`getline`函数、`<FH>`操作符和`chomp`函数会查看这个变量。它的工作类似于`awk`的`RS`变量，如果设置为`null`串，会把一个或多个空行处理为一个记录终止符（不过空行不能包含隐藏的空格或`tab`）。可以把它设置为一个多字符的字符串来匹配多字符终止符，不过不能把它设置为模式，`awk`在某些方面更擅长。

需要说明，如果文件包含连续的空行，将`$/`设置为`"\n\n"`与把它设置为`" "`的含义稍有不同。把它设置为`" "`时，会把两个或多个连续的空行当作一个空行。将它设置为`"\n\n"`则表示Perl会顽固地认为第三个换行符属于下一段。

如果不定义`$/`，会使下一行输入操作将文件的其余部分完全“吞入”，把它们当作一个标量值：

```
undef $/;          # 启用“整个文件”模式
$_ = <FH>;          # 整个文件都在这里
s/\n[ \t]+/ /g;     # 折叠缩进的行
```

如果使用`while (<>)`构造来访问`ARGV`句柄，而且`$/`未定义，那么每个读操作会得到下一个文件：

```
undef $/;
while (<>) {        # $_ 包含整个下一个文件
    ...
}
```

尽管前面使用了`undef`，不过更安全的做法是使用`local`使`$/`未定义：

```
{
    local $/;
    $_ = <FH>;
}
```

如果将`$/`设置为一个引用，指向一个整数或包含整数的标量，或者指向可以转换为一个整数的标量，这会使`getline`和`<FH>`操作读入定长的记录（最大记录大小就是所引用的整数），而不是以某个特定字符串结束的变长记录。所以以下代码：

```
$/ = \32768; #或"\32768"或$scalar_var_containing_32768
open(FILE, $myfile);
$record = <FILE>;
```

会从`FILE`句柄读入一个不超过32768字节的记录。如果不是读取一个面向记录的文件（或者你的操作系统没有面向记录的文件），那么每个读操作都会得到一整块数据。如果记录长度大于你设置的记录大小，则会分部分地得到这个记录。只有在标准I/O提供了一个`read(3)`函数的系统上，记录模式才能与行模式很好地结合；特别需要指出VMS是一个例外。

`$/`设置为启用记录模式时（或者如果`$/`未定义），调用`chomp`没有任何效果。参见第17章中的`-0`（数字）和`-l`（字母）命令行开关（助记方法：引用诗歌时用`/`分隔各行）。



## @ISA

[PKG] 这个数组包含一些包的名字，如果在当前包中无法找到一个方法调用，就会查看这些包。也就是说，它包含了包的基类。base pragma会隐式地设置这个变量。如果声明了strict，使用这个变量会发出警告。参见第12章。

## @LAST\_MATCH\_END

@+ [DYN, RO] 这个数组包含当前活动动态作用域中最后成功的子匹配末尾的偏移量。 $\$+[0]$ 是整个匹配末尾的偏移量。这与在匹配变量上调用pos函数时返回的值是一样的（我们谈到“末尾的偏移量”时，实际上是指所匹配的字符串末尾后面的第一个字符的偏移量，因此可以从末尾偏移量减去起始偏移量来得到长度）。这个数组的第 $n$ 个元素包含第 $n$ 个子匹配的偏移量，所以 $\$+[1]$ 就是 $\$1$ 末尾的偏移量， $\$+[2]$ 是 $\$2$ 末尾的偏移量，依此类推。可以使用 $\$#+$ 来确定最后一个成功匹配中有多少个子组。参见@- (@LAST\_MATCH\_START)。

与变量\$var成功匹配后：

- $\$`$  等同于substr(\$var, 0,  $\$-[0]$ )
- $\$&$  等同于substr(\$var,  $\$-[0]$ ,  $\$+[0] - \$-[0]$ )
- $\$'$  等同于substr(\$var,  $\$+[0]$ )
- $\$1$  等同于substr(\$var,  $\$-[1]$ ,  $\$+[1] - \$-[1]$ )
- $\$2$  等同于substr(\$var,  $\$-[2]$ ,  $\$+[2] - \$-[2]$ )
- $\$3$  等同于substr(\$var,  $\$-[3]$ ,  $\$+[3] - \$-[3]$ )等

## @LAST\_MATCH\_START

@- [DYN, RO] 这个数组包含当前活动动态作用域中最后成功的子匹配起始位置的偏移量。 $\$-[0]$ 是整个匹配起始位置的偏移量。这个数组中的第 $n$ 个元素包含第 $n$ 个子匹配的偏移量，所以 $\$-[1]$ 就是 $\$1$ 起始位置的偏移量， $\$-[2]$ 是 $\$2$ 起始位置的偏移量，依此类推。可以使用 $\$#-$ 来确定最后一个成功匹配中有多少个子组。参见@+ (@LAST\_MATCH\_END)。

## \$LAST\_PAREN\_MATCH

$\$+$  [DYN, RO] 这个变量从当前活动动态作用域中最后一个成功的匹配返回最后一个有小括号的子匹配。如果你不知道（或者不关心）哪一组候选模式匹配，这会很有用（助记方法：正向前瞻）。例如：

```
$rev = $+ if /Version: (.*)|Revision: (.*)/;
```

## %LAST\_PAREN\_MATCH

%+ [DYN, RO] 类似于%-, 这个变量允许访问当前活动动态作用域中最后一个成功的模式匹配中的命名捕获组。键是捕获组的名字，值是与该名匹配的字符串，或者如

果同一个名字匹配多个组，则是最后一个匹配的组。使用`%-`可以找到所有这些捕获组。

对这个散列使用`each`调用的同时不要在循环本身完成模式匹配，否则结果会出乎你的意料。

如果你不喜欢写为类似`#{NAME}`的形式，可以使用标准`Tie::Hash::NamedCapture`模块为`%+`提供你自己选择的一个别名。

#### `$LAST_REGEXP_CODE_RESULT`

`$_R [DYN]` 这会包含一个成功的模式中用`(?{ CODE })`构造执行的最后一个代码段的结果。`$_R`提供了一种执行代码的方法，并且可以记住结果以备以后在模式中使用。

Perl正则表达式引擎处理模式时，可能会遇到多个`(?{ CODE })`表达式。如果是这样，它会记住各个`$_R`值，如果以后必须回溯返回一个表达式，则会恢复`$_R`原先的值。换句话说，`$_R`在模式中有一个动态作用域，有点像`$1`之类的变量。

所以`$_R`不只是一个模式中执行的最后一个代码段的结果。它是导致一个成功匹配的最后一个代码段的结果。可以得到这样一个推论：如果匹配不成功，`$_R`将恢复到发生这个匹配之前的值。

如果`(?{ CODE })`模式直接作为一个`(?(COND)IFTRUE|IFFALSE)`子模式的条件，则不会设置`$_R`。

#### `$LAST_SUBMATCH_RESULT`

`$_N` 最后一个成功的搜索模式中最近结束的已用组（也就是说，有最右结束小括号的组）所匹配的文本。这主要在`(?...)`块内部用来检查刚匹配的文本。例如，要有效地将文本捕获到一个变量（除了`$1`、`$2`等），可以把`(...)`替换为：

```
(?:(PATTERN)(?{ $var = $_N })))
```

以这种方式设置并使用`$var`，就不用担心它们是哪一组小括号了。

这个变量是v5.8新增的。

助记方法：最近结束的（可能）嵌套的小括号。

#### `$LIST_SEPARATOR`

`$" [ALL]` 一个数组或片段内插到一个双引号字符串（或类似字符串）时，这个变量指定了在各个元素之间放置的字符串。默认为空格（助记方法：显然，大家希望如此）。

`$_M [ALL]` 默认地，内存耗尽是不可捕获的。不过，如果你的Perl编译为可以充分利用`$_M`，可以把它用作为一个紧急内存池。如果编译Perl时提供了`-DPERL_EMERGENCY_SBRK`选项，并使用Perl的`malloc`，则：

```
$_M = "a" x (1 << 16);
```



会分配一个64K的缓冲区在紧急情况下使用。参见Perl源代码版本目录中的`INSTALL`文件，从中可以得到如何启用这个选项的信息。为了防止轻率地使用这个高级特性，这个变量没有`use English`长名（我们也不会告诉你助记方法）。

#### `$MATCH`

`$&` [DYN, RO] 当前活动动态作用域中最后一个成功的模式匹配所匹配的字符串（助记方法：类似于某些编辑器中的`&`）。

如果在程序中随意使用这个变量，会使所有正则表达式匹配的性能大幅下降。为了避免这种影响，可以使用`@-`抽取出相同的子串。从v5.10开始，可以使用`/p`匹配标志和`${^MATCH}`变量对特定的匹配操作完成同样的处理。

#### `${^MATCH}`

[DYN, RO] 这个变量类似于`$&` (`$MATCH`)，不过它不会带来与那个变量相关的性能开销，而且只保证带`/p`修饰符编译或执行模式时才包含一个已定义的值。

这个变量是v5.10新增的。

#### `$OSNAME`

`$^O` [ALL] 指定要为哪些平台（通常是操作系统）编译当前的`perl`二进制版本，这个变量包含了这些平台的名称。与从`Config`模块得到这个信息相比，这种方法要简洁得多。

#### `$OS_ERROR`

##### `$ERRNO`

`$!` [ALL] 如果在数值上下文中使用，会得到最后一个系统调用错误的当前值，以及所有常见的注意事项（这说明不要指望`$!`的值是某个特殊值并依赖这一点，除非得到一个指示系统错误的特定返回值）。如果在字符串上下文中使用，`$!`会得到相应的系统错误字符串。有些情况下可以为`$!`赋一个错误号，例如，你希望`$!`返回一个特定错误的字符串，或者希望为`die`设置退出值，就可以为它设置一个错误号。参见`Errno`（助记方法：刚才是什么错误）。

#### `%OS_ERROR`

##### `%ERRNO`

`%!` [ALL] 仅当`$!`设置为该值时，`%!`的每个元素都有一个`true`值。例如，当且仅当`$!`的当前值是`ENOENT`时，`${!ENOENT}`为`true`；也就是说，如果最近的错误是“没有该文件或目录”（或者相应的错误：并不是所有操作系统都会给出这个错误，当然也不是所有语言都会给出这个错误）。要检查一个特定的键在你的系统上是否有意义，可以使用`exists ${!SOMEKEY}`；要检查一个合法键列表，可以使用`keys %!`。有关的更多信息参见`Errno`模块的文档，另外参见上面的`$!`。

这个变量是v5.005新增的。

[ALL] 只有已经加载了标准`Errno`模块时才会定义这个散列。一旦加载这个模块，可



以使用一个特定的错误字符串指定%!的下标，它是当前错误时%!的值才为true。例如，只有当C的errno当前设置为#define值ENOENT时，\$!{ENOENT}才为true。这对于访问开发商特定的符号很方便。

autoflush HANDLE EXPR

\$AUTOFLUSH

\$| [FHA] 如果设置为true，对于当前选择的输出句柄，每个print、printf和write之后都会强制刷新输出缓冲区（我们把这个命令称为缓冲（*buffering*）。与通常的看法不同，设置这个变量并不是关闭缓冲）。这个变量默认为false，在很多系统上，这表示如果输出到终端，那么STDOUT是行缓冲，否则为块缓冲，甚至管道和套接字上也是如此。如果你要输出到一个管道，例如在`rsh(1)`下运行一个Perl脚本并且希望看到运行时得到的输出，设置这个变量就很有用。这个变量设置为true时，如果当前选择的文件句柄的输出缓冲区还有未处理、未刷新输出的数据，作为赋值的一个副作用，会立即刷新输出这个缓冲区。参见单参数形式的select，其中给出的例子展示了对于STDOUT以外的文件句柄如何控制缓冲（助记方法：希望你的管道滚烫）。

这个变量对输入缓冲没有任何影响，要控制输入缓冲，参见第27章的getc，或者参见POSIX模块中的例子。

\$OUTPUT\_FIELD\_SEPARATOR

\$/ [ALL] print的输出字段分隔符（实际上是终止符）。正常情况下，print只打印你指定的列表元素，在它们之间不加任何其他内容。通过设置这个变量（就像设置awk的OFS变量），可以指定字段之间要打印哪些内容（助记方法：print语句中有一个“,”时要打印什么）。

\$OUTPUT\_RECORD\_SEPARATOR

\$\ [ALL] print的输出记录分隔符（实际上是终止符）。正常情况下，print只打印你指定的逗号分隔的字段，没有末尾的换行符或记录分隔符。通过设置这个变量（就像设置awk的ORS变量），可以指定print末尾打印什么（助记方法：设置\$\而不是在print末尾增加“\n”。另外，它与/很类似，不过这是从Perl“拿回来”的）。参见第17章中的-l（表示“line”）命令行开关。

%OVERLOAD

[NOT, PKG] 这个散列的记录由use overload pragma在内部设置，为当前包的类对象实现操作符重载。参见第13章。

\$PERLDB

^P [NOT, ALL] 启用Perl调试器（*Perl-d*）的内部变量。

\$PERL\_VERSION

^V [ALL] Perl解释器的修订版、版本和子版本。这个变量最早出现在v5.6.0中；对于之前的perl版本，只会看到一个未定义的值。在v5.10.0之前，\$^V表示为一个v-string

(版本字符串)。

`$^V`可以用来确定执行一个脚本的Perl解释器是否在正确的版本范围内。例如：

```
warn "Hashes not randomized!\n" unless $^V && $^V gt v5.8;
```

要把`$^V`转换为它的字符串表示，可以使用`sprintf`的`"%vd"`完成转换：

```
printf "version is v%vd\n", $^V; # Perl的版本
```

Perl的更新版本会自动完成这个转换：

```
$ Perl-E 'say $^V'
v5.14.0
```

```
$ Perl-E 'say $^V 5.10.1'
1
```

参见`use VERSION`和`require VERSION`的文档，可以了解到如果运行的Perl解释器比你希望的版本老，可以采用一种很方便的方法失败。参见`$]`了解Perl版本原先的表示。

助记方法：使用`^V`完成版本控制。

## `$POSTMATCH`

`$'` [DYN, RO] 当前活动的动态作用域中最后一个成功模式所匹配内容后面的字符串（助记方法：`'`通常在一个引号字符串后面）。例如：

```
$_ = "abcdefghi";
/def/;
print "$`:$&:$'\n"; # 打印abc:def:ghi
```

由于是动态作用域，Perl无法知道哪些模式需要把它们的结果保存到这些变量中，所以程序中只要有`$``或`$'`就会导致整个程序中的所有模式匹配性能降低。对于小程序来说这不算大问题，但是如果你要写可重用的模块代码，就应当避免使用`$``和`$'`。

上面的例子可以重写为以下等价的代码，不过这里不会导致全局性能下降：

```
$_ = "abcdefghi";
/(.*?)(def)(.*)/s;      # /s以防$1包含换行符
print "$1:$2:$3\n";      # 打印abc:def:ghi
```

## `${^POSTMATCH}`

[DYN, RO] 这个变量类似于`$'` (`$POSTMATCH`)，不过不会像`$'`那样导致模式匹配性能下降，而且只保证带`/p`修饰符编译或执行模式时才包含一个已定义的值。

这个变量是v5.10新增的。

## `$PREMATCH`

`$`` [DYN, RO] 当前活动的动态作用域中最后一个成功模式所匹配内容前面的字符串（助记方法：```通常在一个引号字符串前面）。参见前面`$'`中关于性能的介绍。

## `${^PREMATCH}`

这个变量类似于`$`` (`$PREMATCH`)，不过不会像`$``那样导致模式匹配性能下降，而且



只保证带/p修饰符编译或执行模式时才包含一个已定义的值。

这个变量是v5.10新增的。

#### `$PROCESS_ID`

`$$` [ALL] 运行这个脚本的Perl的进程号 (PID)。调用fork时这个变量会自动更新。实际上, 你甚至可以自己设置`$$`; 不过, 这不会改变你的PID。改变PID有一个很巧妙的技巧 (助记方法: 与各个shell中的做法一样)。

需要当心, 如果会被错误地解释为一个解引用`$$alphanum`, 就不要使用`$$`。对于这种情况, 应当写为`${$}alphanum`以区别`${$alphanum}`。

#### `$PROGRAM_NAME`

`$0` [ALL] 包含正在执行的Perl脚本文件的名称。赋值为`$0`是有魔法的: 它会尝试修改`ps(1)`程序通常报告的参数区。这个变量可以用来隐藏你正在运行的程序, 但是作为一种指示当前程序状态的方法则更为有用。不过并非所有系统上都可用 (助记方法: 与`sh`、`ksh`、`bash`等中相同)。

在多线程脚本中, Perl会协调这些线程, 使得任何线程都可以修改其`$0`副本, 而且这个改变对`ps`是可见的 (假设操作系统支持这个特性)。需要说明, 其他线程看到的`$0`不会改变, 因为它们都有自己的副本。

如果通过开关`-e`或`-E`向Perl提供程序, `$0`将包含字符串`"-e"`。

#### `$REAL_GROUP_ID`

`$(` [ALL] 这个进程的真实组ID (GID)。如果你所在的平台支持同时属于多个组, `$(`会提供你所在的组列表, 各个组之间用空格分隔。第一个数是`getegid(2)`返回的组号, 后面是`getgroups(2)`返回的组号, 其中某个数可能与第一个数相同。

不过, 赋给`$(`的值必须是一个数, 用来设置真实GID。所以`$(` (给定的值在强制转换为数字之前不能赋回给`$(`, 例如可以通过加`0`将它转换为一个数。这是因为, 你可能只有一个真实组。参见`$(` (`$EFFECTIVE_GROUP_ID`), 它允许设置多个有效组。

助记方法: 小括号用于完成分组。如果运行`setgid`, 真实GID是你离开的组。

#### `$REAL_USER_ID`

`$<` [ALL] 这个进程的真实用户ID (UID), 类似于`getuid(2)`系统调用返回的用户ID。是否修改这个变量以及如何修改取决于系统的不同实现, 参见`$>` (`$EFFECTIVE_USER_ID`) 下的例子。因为对`$<`的修改需要一个系统调用, 所以在尝试修改之后要检查`$!`, 以检测可能的错误 (助记方法: 如果运行`setuid`, 说明你来自这个UID)。

#### `%SIG`

[ALL] 这个散列用来为各个信号设置信号处理器。(参见第15章中的“信号”一节)。例如:



```

sub handler {
    my $sig = shift;      # 第1个参数是信号名
    syswrite STDERR, "Caught a SIG$sig--shutting down\n";
                          # 避免异步处理中的标准I/O
                          # 以避免内核转储（即使
                          # 字符串连接有风险）。
    close LOG;           # 这会调用标准I/O，所以
                          # 可能转储内核！
    exit 1;              # 不过由于我们正要退出，
                          # 所以不会带来危害
}

$SIG{INT} = \&handler;
$SIG{QUIT} = \&handler;
...
$SIG{INT} = "DEFAULT";   # 恢复默认动作
$SIG{QUIT} = "IGNORE";   # 忽略SIGQUIT

```

%SIG散列包含未定义的值，对应于未设置处理器的信号。处理器可以指定为一个子例程引用或者指定为一个字符串。如果指定字符串值，只要不是两个特定动作“DEFAULT”或“IGNORE”，则是一个函数名，如果没有用包限定，就会解释为在main包中。下面再给出一些例子：

```

$SIG{PIPE} = "Plumber";  # 可以，假设为main::Plumber
$SIG{PIPE} = \&Plumber;  # 可以，使用当前包的Plumber

```

还可以使用%SIG散列设置一些内部钩子。要打印一个警告消息，会调用\$SIG{\_\_WARN\_\_}指示的子例程。警告消息会作为第一个参数传递。如果有\_\_WARN\_\_钩子，则不会以通常的方式向STDERR打印警告。可以使用这个特性把警告保存在一个变量中，或者把警告转变为致命错误，如下所示：

```

local $SIG{__WARN__} = sub { die $_[0] };
eval $proggie;

```

这类似于：

```

use warnings qw/FATAL all/;
eval $proggie;

```

只不过第一个有动态作用域，而第二个有词法作用域。

\$SIG{\_\_DIE\_\_}指示的例程提供了一种方法，一个有魔法的吻可以把“青蛙”异常变成一个“王子”异常，而通常这是不行的。最好的用法是，如果一个程序由于未捕获的异常而将要终止，可以在最后时刻完成一些处理。尽管不能完全自救，不过总能做最后的挣扎。

异常消息会作为第一个参数传递。\_\_DIE\_\_钩子例程返回时，异常处理仍继续，就像没有钩子一样，除非钩子例程本身通过一个goto、一个循环退出或die退出。在这个调用中要显式禁用\_\_DIE\_\_处理器，使你能自己从\_\_DIE\_\_处理器调用真正的die（如果没有禁用，\_\_DIE\_\_处理器就会永远递归地调用自己）。\$SIG{\_\_WARN\_\_}的

处理器也有类似的做法。

只有主程序可以设置\$SIG{\_\_DIE\_\_}，而模块不能。这是因为，目前即使异常被捕获，也会触发\$SIG{\_\_DIE\_\_}处理器。强烈建议不要设置\$SIG{\_\_DIE\_\_}，因为这有可能破坏一些无辜的模块，它们并不希望预期的异常被莫名其妙地修改。只能把这个特性作为最后一着，而且如果必须使用，那么一定要在前面加上local来限制这个危险的时期。

对未捕获的异常做出响应时，有些编程语言会提供一个杂乱的堆栈转存，统统显示在屏幕上，如果你习惯这些编程语言的做法，可以让Perl做相同的事情，只需要在主程序中加入以下代码：

```
use Carp;
$SIG{__DIE__} = sub { confess "$0: UNCAUGHT EXCEPTION: @_ " unless $^S };
```

不要试图利用这个特性建立一个异常处理机制。应当使用eval {}捕获异常。例如，不要使用一个\_\_DIE\_\_钩子，更简洁的做法是将整个主程序都放在一个子例程中，并用一个标准异常捕获代码包围这个子例程——作为一个常规的eval BLOCK：

```
use Carp;
eval {
    function_that_does_everything();
    1;
} || confess "$0: Caught unexpected exception: $@";
```

STDERR

[ALL] 任何包中标准错误的特殊文件句柄。

STDIN

[ALL] 任何包中标准输入的特殊文件句柄。

STDOUT

[ALL] 任何包中标准输出的特殊文件句柄。

\$SUBSCRIPT\_SEPARATOR

\$; [ALL] 用于多维散列模拟的下标分隔符。如果一个散列元素表示为：

```
$foo{$a,$b,$c}
```

这实际上表示：

```
$foo{join($;, $a, $b, $c)}
```

不过不要写为：

```
@foo{$a,$b,$c} # 一个片段，注意@
```

这表示：

```
($foo{$a},$foo{$b},$foo{$c})
```

默认为"\034"，等同于`awk`中的SUBSEP。需要说明，如果键包含二进制数据，可能没有安全的\$;值（助记方法：逗号——语法中的下标分隔符，是半个分号。没错，我们知道这很牵强，不过\$,有更重要的用途）。

尽管我们没有废弃这个特性，不过你现在应该考虑使用“真正的”多维散列，如`$foo{$a}{$b}{$c}`，而不是`$foo{$a,$b,$c}`。但模拟的多维散列可能更容易排序，而且更适合用作为简单的DBM文件。

## `$SYSTEM_FD_MAX`

`$^F` [ALL] 最大“系统”文件描述符，正常情况下为2。系统文件描述符会在`exec`执行过程中传递到新程序，但更高的文件描述符不能传递。另外，在`open`期间，系统文件描述符会保留（即使`open`失败）。通常在尝试`open`之前会关闭文件描述符，如果`open`失败则保持关闭。需要说明，文件描述符的“执行时关闭”状态将根据`open`时（而不是`exec`时）的`$^F`值决定。可以临时提升`$^F`来避免这个问题：

```
{
    local $^F = 10_000;
    pipe(HITHER,THITHER) || die "can't pipe: $!";
}
```

## `${^TAINT}`

[ALL, RO] 这个只读变量反映了污染模式是打开、关闭还是只提供警告：

0	污染模式关闭（默认）。
1	污染模式打开，通常因为程序是用 <code>-T</code> 命令行开关运行的。
-1	只提供污染警告，由 <code>-t</code> 或 <code>-TU</code> 命令行开关启用。

这个变量是v5.8新增的。

## `${^UNICODE}`

[XXX, ALL] 这个变量反映Perl的某些内部Unicode设置。这个变量在Perl启动时由`-C`命令行开关或`PERL_UNICODE`环境变量设置为某个数字值；在此之后，它是只读的。

这个变量是v5.8.2新增的。

## `${^UTF8CACHE}`

[NOT, ALL] 这个内部变量控制内部UTF-8偏移量缓存代码的状态：

1	打开（默认）
0	关闭
-1	调试缓存代码，为此要检查线性扫描的结果并警告差异。由 <code>-Ca</code> 命令行开关设置



这个变量是v5.8.9新增的。

#### `${^UTF8LOCALE}`

[NOT, ALL] 这个变量指示启动时Perl是否检测UTF-8本地化环境。Perl在其“调整utf8为本地化环境”模式时会由`-CL`命令行开关设置这个信息。

这个变量是v5.8.8新增的。

#### `$VERSION`

[PKG] 指定一个模块的最小可接受版本时，就会访问这个变量，如`use SomeMod 2.5`。如果`$SomeMod::VERSION`小于这个最小可接受的版本号，则会产生一个异常。理论上讲，应当由`UNIVERSAL->VERSION`方法查看这个变量，所以如果你希望有非默认的其他行为，可以在当前包中定义你自己的`VERSION`函数。参见第12章。

#### `$WARNING`

`$^W` [ALL] 全局警告开关的当前布尔值（不要与全球变暖混淆，关于这一点我们已经听到很多全球范围的警告）。参见第29章中的`warnings pragma`，以及提供词法作用域警告的`-W`和`-X`命令行开关，它们不会受这个变量的影响（助记方法：这个值与`-w`开关有关）。

#### `${^WARNING_BITS}`

[NOT, ALL] `use warnings pragma`启用的警告检查的当前设置。更多详细信息参见第29章的`use warnings`。

#### `${^WIDE_SYSTEM_CALLS}`

[ALL] 这是一个全局标志，允许Perl做出的所有系统调用使用系统自身的宽字符API（如果提供了这些API）。还可以从命令行使用`-C`命令行开关来启用。初始值通常是0，以便与v5.6之前的Perl版本兼容，不过如果系统提供了一个用户可设置的默认值（如通过`$ENV{LC_CTYPE}`），Perl可能会自动将它设置为1。`bytes pragma`总会在当前词法作用域中覆盖这个标志的作用。

#### `${^WIN32_SLOPPY_STAT}`

[ALL] 如果这个变量设置为一个true值，则在Windows上`stat`不会尝试打开这个文件。这意味着链接数无法确定，如果存在这个文件的其他硬链接，文件属性可能过时。另一方面，由于不打开文件，这样速度会更快，特别是网络驱动器上的文件。

这个变量可以设置在`sitecustomize.pl`文件中，将本地Perl安装配置为默认使用“sloppy”`stat`。参见`perlrun`“命令开关”中有关`-f`的文档，了解网站定制的更多信息。

这个变量是v5.10新增的。

Perl能够以多种不同方式拆分文本，这一点让它颇具盛名，这正是Perl名字（Perl全名为实用摘录与报表语言）中的摘录（Extraction）部分，Perl能够如此流行这一部分绝对功不可没。利用Perl还可以很容易地创建格式化字符串来建立报表（Report）。这一章将介绍printf和sprintf函数、pack和unpack函数，以及格式，格式原先在行式打印机上用来打印格式美观的报表，不过进入新千年后有时也很有用。

## 字符串格式

Perl可以按C的库函数*sprintf*常用的printf约定创建格式化字符串。*sprintf*版本返回一个字符串，printf版本可以输出到默认文件句柄或指定的文件句柄：

```
sprintf FORMAT, LIST
printf FORMAT, LIST
printf FILEHANDLE FORMAT, LIST
```

*sprintf*参数处理稍有点特殊。它的第一个参数总是作为一个标量，即使它实际上是一个数组。这可能不是你想要的，因为它会在标量上下文中使用@array，只打印数组的元素个数：

```
my @array = ( '%d %d %d', 1, 2, 3 );
sprintf @array;
```

printf的参数有所不同，它要处理一个可选的FILEHANDLE参数。

FORMAT字符串是包含内嵌指示符的文本，这些指示符会替换为LIST的元素，每个字段分别替换为列表中的一个元素。这是Perl从C“窃取”的特性之一，所以可以查看你的系统上的*sprintf(3)*或*printf(3)*来了解一般原则的有关解释。

Perl会完成自己的sprintf格式化，它模拟了C函数*sprintf*，不过并没有直接使用这个函数<sup>注1</sup>。因此，你的本地*sprintf(3)*函数中的任何非标准扩展在Perl中都不可用。

Perl的*sprintf*支持公认常用的一些转换，如表26-1所示。

表26-1：Sprintf的格式

字段	含义
%%	直接量百分号
%b	无符号整数，二进制
%B	类似于%b，不过通过#flag使用一个大写的“B”
%c	指定序数值的字符
%d	有符号整数，十进制
%e	浮点数，采用科学计数法，使用一个小写的“e”
%E	类似于%e，不过使用一个大写的“E”
%f	浮点数，采用固定十进制记法
%g	浮点数，采用%e或%f记法
%G	类似于%g，不过使用一个大写的“E”（如果可用）
%h	C short或无符号short，取决于构建 <i>perl</i> 的编译器
%n	将目前为止输出的C char个数存储到下一个参数中
%o	无符号整数，八进制
%p	指针（采用十六进制输出Perl值的地址）
%s	未指定宽度的一个字符串
%u	无符号整数，十进制
%x	无符号整数，十六进制，使用小写字母
%X	无符号整数，十六进制，使用大写字母

从表26-2可以了解不同值类型可能完成的转换。

表26-2：值类型对应的格式

类型	格式
整数	%b %B %d %h %o %p %u
浮点数	%e %E %f %g %G
字符串	%c %s

注1： 但浮点数除外，对于浮点数只允许标准修饰符。



对于一些数值转换，可以指定printf如何解释这些数字，而不是依赖编译器提供的大小。参见表26-3。

表26-3: printf数值转换

字段	含义
hh	C char或unsigned char (v5.14及以后版本)
h	C short或unsigned short (v5.14及以后版本)
j	C 类型intmax_t (v5.14或以后版本, 带C99编译器)
l	C long或unsigned long
q, L, ll	C long long, unsigned long long或quad (编译器必须支持quad)
t	C ptrdiff_t (v5.14或以后版本)
v	将字符串解释为整数向量，输出为用点号或任意字符串（标志前面有*时从参数表接收的字符串）分隔的数字
z	C size_t (v5.14或以后版本)

要保证向后（也表示“反向”）兼容，Perl还允许表26-4所示的这些不必要的转换，不过它们得到了广泛支持。我们把这些转换从表26-1中分出来就是希望你不要使用这些转换。

表26-4: 数值转换的向后兼容同义词

字段	含义
%i	%d的同义词
%D	%ld的同义词
%U	%lu的同义词
%O	%lo的同义词
%F	%f的同义词

%和格式字母之间可以指定多个额外的属性，来控制格式的解释，如表26-5所列。

表26-5: printf的格式修饰符

标志	含义
space	在正数前加一个空格前缀
+	在正数前加一个加号前缀
-	在字段中左对齐
0	使用0而不是空格来完成右对齐
#	用“o”作为非0八进制前缀，用“0x”作为十六进制前缀

表26-5: sprintf的格式修饰符 (续)

标志	含义
*	使用下一个参数的值作为字段宽度
<i>number</i> \$	使用位置 <i>number</i> 上的参数值
* <i>number</i> \$	使用位置 <i>number</i> 上的参数值作为字段宽度
<i>number</i>	最小字段宽度 (没有相应的最大字段宽度)
. <i>number</i>	“精度”：对于浮点数，这是小数点之后的位数，对于字符串，这将是最大长度，对于整数，精度则是最小长度

下面给出一些例子。在指示符前面加一个空格会在数字前加一个空格，而不论数字大小：

```
printf "<% d>", 1; # "< 1>"
```

使用一个+为数字追加一个正号，即使数字是0<sup>注2</sup>：

```
printf "<%+d>", 1; # "<+1>"
printf "<%+d>", 2; # "<+2>"
```

结合使用一个空格和一个+（顺序为先空格后+），会在正数前放一个+：

```
printf "<%+ d>", 3; # "<+3>"
printf "<% +d>", 5; # "<+5>"
```

为一个整数指定精度会为它填充0。+不计入精度：

```
printf "<%.5d>", 8; # "<00008>"
printf "<%+.5d>", 13; # "<+00013>"
```

不过，如果精度宽度大于实际宽度，会填充0以达到精度宽度。可以对值左对齐或右对齐：

```
printf "<%-10.6d>", 21; # "<000021 >"
printf "<%10.6d>", 34; # "< 000034>"
printf "<%010.6d>", 55; # "< 000055>"
printf "<%+10.6d>", 89; # "< +000089>"
```

这适用于所有整数格式。

字符串默认地右对齐，不过如果有一个负号，会让字符串左对齐：

```
printf "<%6s>", 144; # "< 144>"
printf "<%-6s>", 233; # "<233 >"
```

前导0会用0填充空位置，不过只填充左边，而且即使值不是一个数字也会在左边填充0：

注2： 这与0+和0-的概念不同，这两个分别是从左右两边接近0的极限。

```
printf "<%06s>", 377;    # "<000377>"
printf "<%-06s>", 610;   # "<610 >" - 没有在右边填充0
printf "<%06s>", "Perl"; # "<00Perl>"
```

要采用一个固定列格式对齐字符串，使用宽度会很方便。不过，`printf`只使用最小宽度。它不会截断字符串：

```
printf "<%5s>", "Amelia"; # "<Amelia>", 包括所有6个字符
```

如果你想截断字符串以防溢出，可以在字段宽度后面使用`.number`：

```
printf "<%5.5s>", "Camelia"; # "<Camel>", 只有5个字符
```

字段宽度与你获取的字符数并不一定匹配。如果字符数多于宽度，字符串仍会溢出：

```
printf "<%3.5s>\n", "Camelia"; # "<Camel>"
```

正常情况下，`printf`会用下一个未用的参数填入指示符，不过可以用`number$`明确地告诉它要使用哪个参数。使用`number$`时，要注意应当使用未内插的字符串，或者要对美元符转义，否则Perl会认为你希望在这里内插一个变量。

```
printf '%2$d %1$d', 12, 34;    # "34 12"
printf '%3$d %d %1$d', 1, 2, 3; # "3 1 1"
```

另外允许重用参数：

```
printf '%2$d %1$d %2$d', 12, 34; # "34 12 34"
```

有时你无法提前知道宽度，所以可以把它指定为一个参数，并用\*从下一个参数取得这个值。这会在字符串参数前取一个宽度参数：

```
printf "<%*s>", 6, "Perl"; # "< Perl>"
```

如果参数的值是负值，则左对齐：

```
printf "<%*s>", -6, "Perl"; # "<Perl>"
```

如果你不想使用下一个参数，可以使用`number$`按位置指定宽度参数：

```
printf '<.*2$s>', "a", 6; # "< a>"
```

如果你想从参数表取得字段宽度和字符串的最大字符数，这两个地方都可以使用\*：

```
printf "<%*.*s>\n", 10, 5, "Camelia"; # "< Camel>"
```

不过只能使用一个宽度参数：

```
printf '<.*2$.*s>', "Camelia", 10, 5; # "< Camelia>"
printf '<%*.*2$s>', "Camelia", 10, 5; # "<.*2$s>"
```

#会在数字前增加一个额外的字符表示它的基数，不过前提是值非0（如果值为0，基数是



什么都没有区别)：

```
printf "< %#o>", 37; # "<045>"

printf "< %#x>", 42; # "<0x2a>"
printf "< %#X>", 42; # "<0X2A>"

printf "< %#b>", 137; # "<0b10001001>"
printf "< %#B>", 137; # "<0B10001001>"
```

在%o转换中给定一个#标志和一个精度时，前导“0”不计入精度：

```
printf "< %#.5o>", 0377;      # "<00377>"
printf "< %#.5o>", 010755;    # "<010755>"
printf "< %#.0o>", 0;         # "<0>"
```

对于浮点值(%e、%f和%g)，可以用`number`指定小数点后的小数位数。如果使用了宽度，这要放在宽度后面。注意这个指示符会对数字取整：

```
printf "< %f>", 3.14159265;    # "<3.141593>"
printf "< %.1f>", 3.14159265; # "<3.1>"
printf "< %.0f>", 3.14159265; # "<3>"

printf "< %e>", 6.62606857e-34; # "<6.626069e-34>"
printf "< %.1e>", 1.05457148e-34; # "<1.1e-34>"
```

如果`use locale`（参见第29章）起作用，而且调用了`POSIX::setlocale`，格式会使用对应该本地化环境的十进制分隔符：

```
use POSIX;
use locale;

POSIX::setlocale(LC_NUMERIC, "fr_FR");
printf "< %f>", 3.1415926; # "<3,141593>"
```

%g指示符会使用你的系统首选项，所以你得到的结果可能会稍有不同：

```
printf "< %g>", 1 << 31;      # "<2.14748e+09>"
printf "< %.5g>", 1 << 31;    # "<2.1475e+09>"
printf "< %.10g>", 1 << 31;   # "<2147483648>"
```

小于100的指数所用的位数取决于你的系统；有些可能会用0填充：

```
printf "< %g>", 1 << 31;      #可能是"<2.14748e+009>"
```

v修饰符与其他修饰符不同。它会分解它的字符串参数，认为每个字符是一个整数，并按你的要求格式化。它用一个点号连接这些整数。如果结合一个十六进制格式指示符，采用一种“类Unicode”记法打印码点序列时会很方便：

```
printf "< %vd>", "\x5\xE\x2"; # "5.14.2"

use utf8;
printf "< %vd>", "À%{"; # "<65.758.37.123>"
```

```
printf "<%vX>", "À%{"; # "<41.300.25.7B>"
printf "U+v%04x", "À%{"; # "U+0041.0300.0025.007B"
```

注意上面的字形“À”需要两个码点。

如果你不想点号来连接这些数字，也可以指定你的分隔符作为一个参数：

```
printf "<.*vX>", ":", "À%{"; # "<41:300:325:7B>"
printf "<.*2$vX>", "À%{", ":"; # "<41:300:25:7B>"
```

这里需要注意字形问题（特别是多码点字形簇）。对于本章将要介绍的另外两类格式，Perl在计算包含非打印字符、组合标记和宽字符的Unicode数据的宽度时会给出不正确的答案。计算printf和sprintf的格式字符串中的字段宽度时也存在这个问题。第6章“字形和规范化”一节中给出了一个例子，展示了如何使用Unicode::GCString模块的columns方法哄骗printf在所有这些情况下都能给出正确的答案，尽管单靠printf是不行的。基本策略是：在使用更聪明的columns方法之前，先预填充为正确的宽度，而不是期望“头脑简单”的printf把这么复杂的东西搞清楚。

## 二进制格式

如果你熟悉更传统的一些语言，可能见过记录或struct类型等概念。sprintf主要面向人类可读的输出，与sprintf不同，pack和unpack函数则在底层使用，将基本数据类型重复地转换和格式化为这些结构或记录类型的字符串表示（或者从字符串表示转换回原来的结构或记录类型）。这两个函数有一个共同的模板语言，只是稍有差别，见下一节的介绍。

### pack

```
pack TEMPLATE, LIST
```

这个函数取一个普通的Perl值LIST，根据TEMPLATE将它们转换为一个字节串，并返回这个串。pack函数会根据需要填充或截断参数表。也就是说，如果你提供的参数比TEMPLATE需要的参数少，pack会假设其余参数为null。如果你提供的参数比TEMPLATE需要的参数多，额外的参数会被忽略。TEMPLATE中不可识别的格式元素会产生一个异常。

模板将字符串的结构描述为一个字段序列。每个字段用一个字符来表示，描述这个值的类型及其编码。例如，格式字符N指定采用大端字节顺序的一个无符号4字节整数。

字段按模板中给定的顺序打包。例如，要把一个无符号单字节整数和一个单精度浮点值打包到一个字符串，可以写为：

```
$string = pack("Cf", 244, 3.14);
```

返回的字符串中第一个字节值为244。其余的字节是3.14的编码（作为一个单精度浮点数）。浮点数的具体编码取决于你的计算机的硬件。

打包时要考虑一些重要的问题：

- 数据的类型（如整数、浮点数或字符串）。
- 值的范围（如整数为单字节、两字节、4字节或者可能甚至是8字节，或者打包8位字符还是Unicode字符）。
- 整数有符号还是无符号。
- 要使用的编码（如位和字节的原生（native）、小端（little-endian）或大端（big-endian）打包）。

表26-6、表26-7列出了这些格式字符、修饰符及其含义（格式中还可以出现其他字符，将在后面介绍）。

表26-6：pack/unpack的模板字符

字符	含义
a	用null填充的字节串
A	用空格填充的字节串
b	位串，每个字节中采用升序位序（如vec）
B	位串，每个字节中采用降序位序
c	有符号char值（8位整数）
C	无符号char值（8位整数）。Unicode参见U
d	原生格式的双精度浮点数
D	原生格式的float或双精度浮点数；只有当你的系统支持long double而且编译perl时指定了这一点时才可以使用long double
f	原生格式的单精度浮点数
F	原生格式的Perl内部浮点数（NV）
h	十六进制串，低半字节在前
H	十六进制串，高半字节在前
i	有符号整数值，原生格式；至少为32位，不过取决于你使用的C编译器
I	无符号整数值，原生格式；至少为32位，不过取决于你使用的C编译器
j	Perl内部有符号整数(IV)
J	Perl内部无符号整数(UV)
l	有符号long值，总是32位
L	无符号long值，总是32位
n	采用“网络”（大端）字节顺序的16位short
N	采用“网络”（大端）字节顺序的32位long



表26-6：pack/unpack的模板字符（续）

字符	含义
p	指向一个结构（以null结尾的字符串）的指针
P	指向一个定长字符串的指针
q	有符号quad值（64位整数）
Q	无符号quad值（64位整数）；仅当你的系统支持64位而且编译perl时指定了这一点时才可以使用
s	有符号short值，总是16位
S	无符号short值，总是16位
u	uuencoded字符串
U	Unicode字符数字；这会转换为一个字符模式的字符和一个字节模式的UTF-8编码字符
v	采用“VAX”（小端）顺序的16位short
V	采用“VAX”（小端）顺序的32位long
w	BER压缩的整数
W	无符号char值
x	Null字节（向前跳过一个字节）
X	反向一个字节
Z	null结尾的（和null填充的）字节串
@	null填充到绝对位置
.	null填充或截断到绝对位置
(	开始一个组
)	结束一个组

表26-7：pack/unpack的模板修饰符

修饰符	应用于	效果
!	iIllaS	强制原生大小
!	xX	使用x和X相当于对齐字符
!	nNvV	当作有符号而不是无符号整数
!	@.	打包字符串表示——危险！
>	dDfFiIjJlLpPqQsS	强制大端字节顺序，可以应用于组和子组
<	dDfFiIjJlLpPqQsS	强制小端字节顺序，可以应用于组和子组

可以在TEMPLATE中任意放置空白符和注释。注释以定制的#符号开头，一直到TEMPLATE中

的第一个换行符（如果有）结束。

## 重复

模板中的每个字母后面可以是一个指示数目（count）的数字，解释为一个重复次数或某种长度，这取决于格式。我们先从这些重复的字母说起（打包为字符或数字）：c, C, d, D, f, F, i, I, j, J, l, L, n, N, p, q, Q, s, S, U, v, V, w和W。

这些字段后面的数字表示重复，所以这个字段会在字符串中重复，可以指定多个参数：

```
$out = pack 'C4', 192, 168, 1, 1;      # \xC0\xA8\01\01
```

重复数可以放在中括号中（可选）：

```
$out = pack 'C[4]', 192, 168, 1, 1;    # \xC0\xA8\01\01
```

中括号中如果是另一个字母，会使用相应格式的长度作为重复数：

```
$out = pack 'C[N]', 192, 168, 1, 1;    # \xC0\xA8\01\01  
$out = pack 'C[s]', 192, 168, 1, 1;    # \xC0\xA8
```

如果没有足够的参数，pack会用null填充其余的参数：

```
$out = pack 'C4', 192, 168;            # \xC0\xA8\00\00
```

使用一个\*表示取所有余下的参数：

```
$out = pack 'C*', 192, 168, 1, 1;      # \xC0\xA8\01\01
```

其他字母对于重复有其他含义。a、A或Z后的数字指定了要填充的字段长度：

```
$out = pack 'A10', 192, 168, 1, 1;    # \xC0\xA8\01\01
```

如果这个数字小于字符串宽度，a和A会截断字符串：

```
$out = pack 'A4', 'Amelia';           # "Amel"  
$out = pack 'a4', 'Amelia';           # "Amel"
```

如果有\*，a和A生成一个与其参数长度相同的字段：

```
$out = pack 'A*', 'Amelia';            # "Amelia"  
$out = pack 'a*', 'Amelia';            # "Amelia"
```

不过Z会保留最后一个位置放置末尾的null字节：

```
$out = pack 'Z4', 'Amelia';            # "Ame\000"
```

前提是指定的数字不为0，如果为0，则没有null字节：

```
$out = pack 'Z0', 'Amelia';            # ""
```

<sup>译者注</sup>z取整个字符串，不论它有多长，都以一个null结束：

```
$out = pack 'Z*', 'Amelia'; # "Amelia\000"
```

对于b和B，这个数表示你希望输出的位数。每个b和B只使用输入中各个字符的一位，即最低位，而且每次只设置一位，而不论其长度：

```
$out = pack 'B8', '10011101'; # 0b10011101
$out = pack 'b8', '10011101'; # 0b10111001
```

h和H会做类似的处理，使用这个数作为要生成的半字节个数。不过，这些字母很特殊，因为它们把看上去像是十六进制数的字符解释为数字：

```
$out = pack 'h1', 'a'; # 0x0a
$out = pack 'H1', 'a'; # 0xa0
$out = pack 'H8', 'deadbeef'; # 0xdeadbeef
```

否则，使用低半字节：

```
$out = pack 'h2', '1'; # 0x01
$out = pack 'h2', 'one'; # 0x08
$out = pack 'H2', 'one'; # 0x80
```

\* 结合h或H可以用null填充字符串来得到偶数个半字节：

```
$out = pack 'H*', 'deadbee'; # 0xdeadbee0
```

对于P，这个数指定了要打包的结构的大小。

对于u，这个数是uuencoded编码的字符串的行长度\*。小于3的数（或\*）会处理为45。下面这个格式：

```
$out = pack "u30", $some_string;
```

会取一行：

```
93VYE(')I;F<@=&\@<G5L92!T:&5M(&%L;```
```

不过对于相同的字符串，如果有一个更短的行长度：

```
$out = pack "u15", $some_string;
```

则得到字符串：

```
/3VYE(')I;F<@=&\@<G5L
*92!T:&5M(&%L;```
```

x不消费任何参数，不过它会插入指定数目的null。\*等同于0：

---

译者注：UUencode是在不同平台间发送文件的通用协议，通常用于发送邮件附件。



```
$out = pack "H2 x h2", "dead", "beef"; # 0xde00eb
$out = pack "H2 x3 h2", "dead", "beef"; # 0xde000000eb
$out = pack "H2 x* h2", "dead", "beef"; # 0xdeeb
```

X不消费任何参数。它会回退指定数目的字节（只要不越过目前为止打包的字符串的起始位置）。\*等同于0：

```
$out = pack "H2 X h2", "dead", "beef"; # 0xeb
$out = pack "H2 X3 h2", "dead", "beef"; # 0xde000000eb
$out = pack "H2 X* h2", "dead", "beef"; # 0xdeeb
```

@截断或填充到相对于最内层组的某个位置（或者如果没有组，则相对于整个字符串）。如果目前为止打包的字符串长度大于这个数，则长度缩减为这个数。如果目前为止打包的字符串长度小于这个数，则用null填充。不论哪一种情况，模板的其余部分都会从新位置开始抽取字符串：

```
$out = pack 'A*', 'Amelia'; # Amelia
$out = pack 'A*@3', 'Amelia'; # Ame
$out = pack 'A*@3A*', 'Amelia', 'Camel'; # AmeCamel
$out = pack 'c@5', 137; # 0x8900000000
```

\*等同于0，所以它会截断目前为止完成的所有内容：

```
$out = pack 'A*@*A*', 'Amelia', 'Camel'; # Camel
```

在一个组中，这种截断或填充只应用于这个组：

```
$out = pack 'A(A@4A)A', 'A', 'B', 'C', 'D'; # "AB\000\000\000CD"
$out = pack 'A(A@*A)A', 'A', 'B', 'C', 'D'; # "ACD"
```

.也完成截断或填充，不过它从列表取得这个位置。重复数指定效果从哪里开始：0表示从当前位置开始；如果是一个数，它指定了从哪个组开始，\*指定字符串的开始位置：

```
# 从字符串开始位置截断
$out = pack 'A(A.*A)A', 'A', 'B', 1, 'C', 'D'; # 'ACD'

# 从字符串开始位置填充
$out = pack 'A(A.*A)A', 'A', 'B', 5, 'C', 'D'; # "AB\000\000\000CD"

# 从字符串开始位置截断
$out = pack 'A(A.1A)A', 'A', 'B', 1, 'C', 'D'; # "ABCD"

# 从字符串开始位置填充
$out = pack 'A(A.1A)A', 'A', 'B', 3, 'C', 'D'; # "AB\000\000\000\000CD"

# 从当前位置填充
$out = pack 'A(A.0A)A', 'A', 'B', 0, 'C', 'D'; # "ABCD"
$out = pack 'A(A.0A)A', 'A', 'B', 2, 'C', 'D'; # "AB\000\000CD"
```

## 其他修饰符

/字符允许对字符串打包和解包，打包结构中包含一个字节数，后面跟着字符串本身。可

以写为`length-item/string-item`。`length-item`可以是任意的`pack`模板字母，描述了如何打包长度值。我们用得最多的是整数打包，如`n`（Java字符串）、`w`（ASN.1或SNMP）和`N`（Sun XDR）。`string-item`目前必须是`A*`、`a*`或`Z*`。对于`unpack`，字符串的长度从`length-item`得到，不过如果指定了`*`，则会将其忽略：

```
unpack "C/a", "\04Gurusamy";      # 得到"Guru"
unpack "a3/A* A*", "007 Bond J ";  # 得到 (" Bond","J")
pack "n/a* w/a*", "hello,","world"; # 得到"\000\006hello,\005world"
```

`length-item`不会从`unpack`显式返回。如果为`length-item`字母增加一个`count`，这往往没有太大用处，除非这个字母是`A`、`a`或`Z`。如果`length-item`为`a`或`Z`，打包时可能会引入`null`（`\0`）字符，Perl认为这在数值字符串中是不合法的。

整数格式`s`、`S`、`l`和`L`后面可以跟一个`!`来指示原生`short`或`long`，而不是明确指定为16位或32位。如今，这个问题主要出现在64位平台上，在这些平台上，本地C编译器看到的原生`short`和`long`长度可能并不是16和32（也可以使用`i!`和`I!`，不过这只是为了完整。它们等同于`i`和`I`）。

在构建了Perl的平台上也可以通过`Config`模块得到原生`short`、`int`、`long`和`long long`的实际大小（字节数）：

```
use Config;
say $Config{shortsize};
say $Config{intsize};
say $Config{longsize};
say $Config{longlongsize};
```

`Config`知道`long long`的大小，但这并不意味着你可以使用`q`或`Q`格式（有些系统上确实提供了这些格式，不过你可能可以运行，也可能不能运行）。

长度大于1个字节的整数格式（`s`、`S`、`i`、`I`、`l`和`L`）在不同处理器之间是不可移植的，因为它们采用不同处理器的原生字节顺序。如果你想得到可移植的打包整数，可以使用格式`n`、`N`、`v`和`V`，它们的字节顺序和大小是已知的。

浮点数只采用原生机器格式。因为存在不同种类的浮点格式，另外由于缺乏一种标准的“网络”表示，所以还没有一种实现互换的工具。这意味着，在一个机器上写的打包浮点数据可能在另一个机器上不可读。即使两个机器都使用IEEE浮点数算术运算，这也可能会有问题，因为IEEE规范中并没有明确规定内存表示的字节顺序。

Perl在内部使用`double`来完成所有浮点数计算，所以从`double`转换为`float`再转换回`double`会损失精度。这说明`unpack("f", pack("f", $foo))`并不总能等于`$foo`。

你要负责处理对齐或填充问题，其他程序对此可能会有要求，特别是C编译器创建的那些程序。它们对于如何在当前特定的体系结构中建立一个C `struct`往往有自己特殊的考虑。

打包时你要增加足够的x以应对这个问题。例如，有以下C声明：

```
struct foo {
    unsigned char c;
    float f;
};
```

这可以写为“C x f”格式、“C x3 f”格式或者甚至“f C”格式，还可以是很多其他格式。pack和unpack函数会处理其输入，并输出为扁平的字节序列，因为它们无法知道这些字节到哪里去或从哪里来。

通过对@或.应用!，可以在这些位置使用打包字符串中的字节偏移量。这可能很高效，不过你必须更仔细地考虑字符串，而且要知道它对于其他格式的大小。

<和>用于对d, D, f, F, i, I, j, J, l, L, p, P, q, Q, s和S指示符指定字节顺序（大小端语义）。这些都是打包整数的指示符，还有一些整数指示符已经指定了大小端语义。<强制小端语义，>强制大端语义：

```
$out = pack 'L>', 0xDEADBEEF; # "\xDE\xAD\xBE\xEF"
$out = pack 'L<', 0xDEADBEEF; # "\xEF\xBE\xAD\xDE"
```

## 更多例子

下面再来看一些例子。第一组将一些数值打包为字节：

```
$out = pack "CCCC", 65, 66, 67, 68;    # $out eq "ABCD"
$out = pack "C4", 65, 66, 67, 68;      #作用相同
```

下面对Unicode圆圈字母做同样的处理：

```
$foo = pack("U4",0x24b6,0x24b7,0x24b8,0x24b9);
```

这里也做类似的处理，会加入几个null：

```
$out = pack "CCxxCC", 65, 66, 67, 68; # $out eq "AB\0\0CD"
```

打包short不代表可以移植：

```
$out = pack "s2", 1, 2; # "\1\0\2\0" (小端)
# "\0\1\0\2" (大端)
```

对于二进制和十六进制包，count指示位数或半字节数，而不是生成的字节数：

```
$out = pack "B32", "01010000011001010111001001101100";
$out = pack "H8", "5065726c"; # 都生成"Perl"
```

a字段的长度只应用于一个字符串：

```
$out = pack "a4", "abcd", "x", "y", "z"; # "abcd"
```

要绕过这个限制，可以使用多个指示符：



```
$out = pack "aaaa", "abcd", "x", "y", "z"; # "axyz"
$out = pack "a" x 4, "abcd", "x", "y", "z"; # "axyz"
```

a格式会填充null:

```
$out = pack "a14", "abcdefg"; # "abcdefg\0\0\0\0\0\0\0"
```

这个模板可以打包一个C struct tm记录（至少在某些系统上）：

```
$out = pack "i9pl", gmtime(), $tz, $toff;
```

一般来讲，可以在unpack函数中使用相同的模板，不过有些格式会有不同的表现，特别是a, A和Z。

如果你想把固定宽度的文本字段连接在一起，使用pack时可以结合一个包含多个A或a格式的TEMPLATE：

```
$string = pack("A10" x 10, @data);
```

不要小看“A”：它可以很好地应用于Perl的内部Unicode。不过，它会按码点填充，而不是按逻辑打印列填充。如果需要处理résumé，会得到以下结果：

```
pack("(A10)2", "re\x{301}sme\x{301}", "work")'
"résumé work "
```

不过，如果再来一次，会得到：

```
say pack("(A10)2", "resume", "work")'
resume work
```

参见第6章“字形和规范化”一节中的讨论，了解对于控制字符、组合标记和宽（两列）字符（如很多东亚文字系统中的字符），如何使用Unicode::GCString的columns方法来正确地填充。

如果你想用一个分隔符连接变宽文本字段，可以使用join函数：

```
$string = join("and", @data);
$string = join("", @data); # null分隔符
```

尽管我们的所有例子都使用直接量字符串作为模板，不过你完全可以从磁盘文件导入你的模板。你甚至可以围绕这个函数构建一个完整的关系数据库系统（这会让我们对你刮目相看）。

## unpack

```
unpack TEMPLATE, EXPR
```

这个函数的工作与pack正好相反：它根据TEMPLATE将表示某个数据结构的字符串（EXPR）扩展为一个值列表，并返回这些值。在标量上下文中，这个函数可以用来解包单个值。这

里的`TEMPLATE`与`pack`函数中的格式基本上相同，用于指定要解包的值的顺序和类型。参见`pack`来详细了解有关`TEMPLATE`的介绍。`TEMPLATE`中如果有非法元素，或者如果试图用`x`，`X`或`@`格式移到字符串之外，都会产生一个异常。

字符串会分解为`TEMPLATE`指定的块。每一块分别转换为一个值。一般地，字符串的字节要么是`pack`的结果，要么表示某种C结构。

如果一个字段的重复数大于输入字符串其余部分所允许的长度，这个重复数会悄悄减少（正常情况下，这里都应当使用重复数\*）。如果输入字符串长度大于`TEMPLATE`指定的长度，则忽略字符串的其余部分。

`unpack`函数对于无格式文本数据也很有用，而不只是二进制数据。假设有这样一个数据文件，其中包含类似下面的记录：

2009 The Graveyard Book	Neil Gaiman
2008 The Yiddish Policemen' s Union	Michael Chabon
2007 Rainbows End	Vernor Vinge
2006 Spin	Robert Charles Wilson
2005 Jonathan Strange & Mr Norrell	Susanna Clarke
2004 Paladin of Souls	Lois McMaster Bujold
2003 Hominids	Robert J. Sawyer
2002 American Gods	Neil Gaiman
2001 Harry Potter and Goblet of Fire	J. K. Rowling

这个文件可能由`printf`生成，见本章前面的描述。或者也可能由格式生成，见下一节的介绍。还有可能是外部生成的。不论哪一种情况，都不能使用`split`解析出字段，因为它们没有明确的分隔符。实际上，字段要由它们在记录中的字节偏移量来确定。因此，尽管这是一个常规的文本记录，但由于它采用了一种固定格式，所以你可能希望用`unpack`来进行分解：

```
use v5.14;
while (<>) {
    my($year, $title, $author) = unpack("A4 x A39 A*", $_);
    say "$author won ${year}'s Hugo for $title.";
}
```

这里之所以写为`${year}'s`，是因为Perl会把`$year's`处理为`$year::s`。不过，如果你的源代码中通过`use utf8`来声明使用UTF-8，则可以安全地使用`$year' s`。

除了`pack`中允许的字段，可以用`%number`为字段加前缀，生成各项的一个简单的`number`位累加校验和，而不是各项本身。默认为16位校验和。这个校验和通过累加扩展值的数值来计算得到（对于字符串字段，会取`ord($char)`的和；对于二进制位字段，则是0和1的总和）。例如，下面计算的数与SysV `sum(1)`程序计算的数相同：

```
undef $/;
$checksum = unpack ("%32C*", <>) % 65535;
```

下面会高效地统计一个位串中已设置的位数：

```
$setbits = unpack "%32b*", $selectmask;
```

下面是一个简单的Base64解码器：

```
while (<>) {  
    tr#A-Za-z0-9+/#cd;          # 删除非base64字符  
    tr#A-Za-z0-9+/# -_#;        # 转换为uuencoded编码格式  
    $len = pack("c", 32 + 0.75*length); # 计算长度字节  
    print unpack("u", $len . $_); # uudecode编码并打印  
}
```

p和P格式要小心使用。因为Perl没有办法检查传入unpack的值是否对应于一个有效的内存位置，如果传入一个可能非法的指针值会有灾难性的后果。

如果有更多打包代码，或者如果一个字段或组的重复数大于输入字符串其余部分所允许的长度，结果并没有明确定义：可能减少重复数，或者unpack可能生成空串或0，也可能产生一个异常。如果输入字符串长度大于TEMPLATE所指定的长度，这个输入字符串的其余部分会被忽略。

## 形象格式

Perl提供了一种帮助生成简单报表的机制，你可能经常看到行式打印机打印这种报表（什么？你没有行式打印机）。为了提供方便，Perl可以帮助你设定与打印时外观差不多的输出页面。它可以跟踪一个页面上有多少行、当前页号、何时打印页眉等信息。相关的关键字是从FORTRAN借来的：format用于声明，write用于执行，参见第27章有关的内容。幸运的是，其布局清晰得多，就像BASIC的PRINT USING语句。可以把它想成是一个可怜人的nroff(1)（如果你了解nroff，这听起来也许不像一个建议）。

格式类似于包和子例程，要声明而不是执行，所以它们可以出现在程序中的任何位置（通常最好把所有格式都放在一起）。它们有自己的命名空间，与Perl中所有其他类型在不同的命名空间中。这说明，如果你有一个函数名为“Foo”，它与名为“Foo”的格式并不相同。不过，与一个给定文件句柄关联的格式的默认名字就等于这个文件句柄的名字。因此，STDOUT的默认格式就名为“STDOUT”，而文件句柄TEMP的默认格式也名为“TEMP”。它们看上去是一样的，但实际上不同。

输出记录格式如下声明：

```
format NAME =  
FORMLIST  
.
```

如果忽略NAME，则定义格式STDOUT。FORMLIST包含一个行序列，每一行可以是以下3种类型之一：



- 注释，由第一列中的#指示。
- “形象”行 (picture) 提供一个输出行的格式。
- 参数行，提供插入到前一个形象行中的值。

形象行会按其所见打印，不过某些字段会在行中替换为值<sup>注3</sup>。形象行中的每个替换字段以一个@ (at字符)或^ (脱字符)开头。这些行不会完成任何形式的变量内插。@字段是正常的字段（不要与数组标记@混淆）；另一类^字段则用于完成基本的多行文本块填充。通过在字段中填充多个<、>、|字符来提供字段的长度，它们分别指定左对齐、右对齐或居中。如果变量超出了指定的宽度，则截断。

要注意，一旦在形象行中引入“有趣的”Unicode字符，想得到“有趣的”形象，所有这些关于宽度和对齐的讨论都不再适用。它甚至不能处理非打印ASCII字符。形象格式假设每个码点只占一列。但在Unicode中并不是这样，因为多个码点可能占0个打印列，另外有些码点可能占两个打印列。参见第6章“字形和规范化”一节中的讨论，了解如何使用Unicode::GCString的columns方法来得到Unicode字符串的真正的打印列。

作为右对齐的一种候选形式，还可以使用#字符（在第一个@或^后面）来指定一个数值字段。可以插入一个. 取代某个#字符来对齐小数点。如果为这些字段提供的任何值包含一个换行，则只打印该换行符前面的文本。最后，特殊字段@\*可以用于打印未截断的多行值；@\*通常会单独出现在一个形象行上。

下一行指定的值要与上一个形象行中字段的顺序相同。提供这些值的表达式要用逗号分隔。处理行之前，表达式都在列表上下文中计算，所以一个列表表达式可能生成多个列表元素。如果用大括号包围，表达式可以分解到多行上（如果是这样，开始大括号必须是第一行上的第一个token）。这允许你根据各自的格式字段将值对齐，以便于阅读。

如果一个表达式计算为一个包括小数部分的数，而且相应的形象字段指定这个小数部分应当出现在输出中（这可以是任何形象字段，除非形象字段中只有多个#字符而未内嵌.），用什么字符表示小数点要由当前的LC\_NUMERIC本地化环境确定。这说明，举例来说，如果运行时环境恰好指定为German（德国）本地化环境，则使用逗号而不是点号。有关的更多信息请参见perllocale手册页。

在一个表达式中，一般认为空白符字符\n、\t和\f都等价于单个空格。因此，可以认为这个过滤器会应用到格式中的各个值：

```
$value =~ tr/\n\t\f/ /;
```

---

注3： 不过，这些字段仍保持列的完整性。形象行不会导致字段扩展或收缩，或者来回变换。你看到的列完全是所见即所得的（WYSIWYG），假设你使用的是一种定宽字体。甚至认为控制字符宽度为1。

还有一个空白符字符：`\r`，如果形象行允许，它会强制打印换行符。

以`^`而不是`@`开头的形象行会特殊处理。对于一个`#`字段，如果值未定义，这个字段则取消。对于其他字段类型，脱字符支持一种填充模式。所提供的值不是任意的表达式，而必须是一个标量变量名，其中包含一个文本字符串。Perl会在这个字段中放入尽可能多的文本，然后“砍掉”字符串前面的部分，使得下一次引用这个变量时，可以打印更多的文本（没错，这意味着变量本身会在`write`调用执行期间被修改，原值不再保留。如果你想保留原来的值，可以使用一个便签变量）。正常情况下，要使用一系列垂直对齐的字段来打印一个文本块。你可能希望最后一个字段以文本“...”结束，如果文本太长不能完整显示，输出中就会显示这个“...”。可以修改变量`$`：（如果使用`English`模块，则是`$FORMAT_LINE_BREAK_CHARACTERS`），将它指定为你喜欢的其他字符，则会在这些字符上（或者在它后面）分解字符串。

要知道，这种简化的换行与UAX #14: Unicode Line Breaking Algorithm（Unicode换行算法）所需的复杂换行完全不同。对于Unicode文本，必须使用各个码点的`Line_Break=VALUE`（缩写LB）属性，还要结合一些复杂的表来确定哪里允许换行。这会相当复杂，为了让你有所认识，下面给出`\p{LB=VALUE}`中`VALUE`可取的属性值：

Ambiguous	Contingent_Break	Ideographic	Postfix_Numeric
Alphabetic	Close_Punctuation	Inseparable	Prefix_Numeric
Break_Both	Close_Parenthesis	Infix_Numeric	Quotation
Break_After	Combining_Mark	Line_Feed	Space
Break_Before	Complex_Context	Next_Line	Unknown
Mandatory_Break	Exclamation	Nonstarter	Word_Joiner
Break_Symbols	Glue	Numeric	ZWSpace
Carriage_Return	Hyphen	Open_Punctuation	

正常情况下，如果脚本中没有空白符或短横线，在换行方面尤其有难度，所以这实际上是唯一的方法。CPAN的`Unicode::LineBreak`模块在其发行版中包含了流行的`Unicode::GCString`模块，充分实现了UAX #14，还包括东亚文字系统的处理。你可以利用这个模块来完成更复杂的工作，而不仅限于简化的ASCII。

使用`^`字段可以生成变长的记录。如果需要格式化的文本很短，那么只需用`^`字段将格式化行重复多次。对短数据这样处理时，最后会得到多个空行。要想避免行最后为空，可以在行中任意位置放置一个`~`（波浪线）字符（波浪线本身将在输出时转换为一个空格）。如果在第一个波浪线旁边放上第二个波浪线，这一行会一直重复，直到该行上这个字段中的所有文本都用尽为止（这样是可以的，因为`^`字段会吞掉它们打印的字符串。不过如果使用`@`字段而且有两个波浪线，你提供的表达式最好不要每次都给出相同的值！可以使用一个`shift`或者另外某个操作符（它需要有一个副作用能够用尽这一组值））。

默认地，处理表格顶端的格式与当前文件句柄同名，后面要加上`_TOP`。这个格式会在每个页面的顶端触发。参见第27章中有关`write`的介绍。



下面给出几个例子:

[illegible]

词法作用域变量在格式中不可见，除非格式在该词法作用域变量的作用域中声明。

可以在相同的输出通道上混合使用print和write，不过必须自行处理\$-特殊变量（如果使用English模块则是\$FORMAT LINES LEFT）。

## 格式变量

当前格式名存储在变量\$~ (\$FORMAT\_NAME)中，当前表格顶端格式名存储在\$^ (\$FORMAT\_TOP NAME)中。当前输出页号存储在\$% (\$FORMAT PAGE NUMBER)，另外页面上的行数存储



在`$= ($FORMAT_LINES_PER_PAGE)`中。是否自动刷新输出这个句柄上的输出缓冲区存储在`$| ($OUTPUT_AUTOFLUSH)`中。在每个页面（除了第一个页面）顶端前面输出的字符串存储在`$^L ($FORMAT_FORMFEED)`中。这些变量都对每个文件句柄来设置，所以你需要用`select`选择与格式关联的文件句柄才能影响它的格式变量：

```
select((select(OUTF),
    $~ = "My_Other_Format",
    $^ = "My_Top_Format"
)[0]);
```

很丑，是不是？不过，这是一种很常见的惯用法，所以看到它也不要太奇怪。至少可以使用一个临时变量来保存之前的文件句柄：

```
$ofh = select(OUTF);
$~ = "My_Other_Format";
$^ = "My_Top_Format";
select($ofh);
```

总的来讲，这种方法要好得多，因为不仅可读性有所改善，现在代码中还有一个中间语句，在调试器中单步跟踪时可以在这里停下来。如果使用English模块，甚至可以读变量名：

```
use English;
$ofh = select(OUTF);
$FORMAT_NAME = "My_Other_Format";
$FORMAT_TOP_NAME = "My_Top_Format";
select($ofh);
```

不过这里仍然有那些怪异的select调用。要想避免这些调用，可以使用Perl捆绑发布的IO::Handle模块。现在可以使用小写方法名来访问这些特殊变量：

```
use IO::Handle;
OUTF->format_name("My_Other_Format");
OUTF->format_top_name("My Top Format");
```

这就好多了！

由于形象行后面的值行可能包含任意的表达式（@字段，而不是^字段），可以把更复杂的处理移交给其他函数，如printf或你自己的某个函数。例如，要在一个数字中插入逗号，可以写为：

```
format Ident =  
    @<<<<<<<<<<<<  
    commify($n)  
.
```

要在字段中加入真正的@、~或^，可以这样做：

```
format Ident =
I have an @ here.
```



主要的问题之一。这方面还有待研究<sup>注4</sup>。

这里给出一个策略：如果你有一个固定大小的页脚，可以在每个`write`前检查`$-($FORMAT_LINES_LEFT)`得到页脚，然后在需要时自行打印这个页脚。

还有一种策略：使用`open (MESELF, "|-")`打开一个管道（参见第27章关于`open`的介绍），用`write`写至`MESELF`而不是`STDOUT`。让子进程对`STDIN`完成后处理，用你喜欢的任何方式重新组织页眉和页脚。这种方法不是很方便，不过确实是可行的。

## 访问格式内部细节

要从底层访问内部格式机制，可以使用内置的`formline`操作符直接访问`$^A`（`$ACCUMULATOR`变量）。格式实际上会编译为一系列`formline`调用。例如：

```
$str = formline <<'END', 1,2,3;
@<<< @||| @>>>
END
```

```
say "Wow, I just stored '$^A' in the accumulator!";
```

或者要创建一个`swrite`子例程，它与`write`的关系就像是`sprintf`与`printf`一样，可以这样做：

```
use Carp;
sub swrite {
    croak "usage: swrite PICTURE ARGS" unless @_;
    my $format = shift;
    $^A = "";
    formline($format, @_);
    return $^A;
}

$string = swrite(<<'END', 1, 2, 3);
Check me out
@<<< @||| @>>>
END
print $string;
```

如果使用`IO::Handle`模块，可以如下使用`formline`来包装列72上的文本块：

```
use IO::Handle;
STDOUT->formline("^" . ("<" x 72) . "~~\n", $long_text);
```

接下来是超级庞大的一章，做好准备……

---

注4：当然，不能保证我们一定会解决这个问题。在这个WWW、Unicode、XML、XSLT以及其他更新事物层出不穷的时代，格式已经有点落伍了。



# 函数

这一章按字母顺序<sup>注1</sup>来介绍内置Perl函数，以方便参考。各个函数描述中，首先给出该函数语法的一个简要小结。参数名（如`THIS`）表示具体表达式的占位符，语法概要后面的文本将介绍提供（或忽略）具体参数的语义。

可以认为函数是表达式中的项，类似于直接量和变量。或者可以把它们看作是前缀操作符，能处理它后面的参数。毕竟，大多数时间我们都把它们称为操作符。其中一些操作符，噢，应该说是函数，取一个`LIST`作为参数。`LIST`的元素应当用逗号分隔（或者用`=>`分隔，这实际上也是一种有趣的逗号）。`LIST`的元素在列表上下文中计算，所以每个元素会返回一个标量或列表值，这取决于它对列表上下文的敏感性。返回的各个值，不论是标量还是列表，都将内插为整个标量值序列的一部分。也就是说，全部列表会扁平化为一个列表。从接收参数的函数的角度来看，整个参数`LIST`是一个一维列表值（要想把一个数组内插为单个元素，必须显式地创建并内插这个数组的一个引用）。

使用预定义的Perl函数时，可以在其参数两边加小括号，也可以不加，这一章中的语法概要都省略了小括号。如果你确实要用小括号，对此有一条规则（尽管很简单但有时也会让人有些意外）：如果看起来像一个函数，那么它就是函数，所以优先级并不重要。否则，它就是一个列表操作符或一元操作符，此时优先级就会有影响。要当心，因为即使你在关键字之间增加了空白符和左括号，这也不能阻止它作为一个函数：

```
print 1+2*4;      # 打印9
print(1+2) * 4;   # 打印3!
print (1+2)*4;    # 也打印3!
print +(1+2)*4;   # 打印12
print ((1+2)*4);  # 打印12
```

注1：系统手册页中有时把关系密切的函数放在一起，所以这里也按照这种方式分组。例如，要看`endpwent`的介绍，需要在`getpwn`下查找。

如果运行Perl时提供了-w开关，它会对此做出警告。例如，上面的第二行和第三行会生成类似下面的消息：

```
print (...) interpreted as function at - line 2.  
Useless use of integer multiplication in void context at - line 2.
```

给定函数的简单定义，对于如何传递参数可以有很多不同的观点。例如，使用chmod时，最常用的方法是把文件权限（模式）作为第一个参数传递给这个函数：

```
chmod 0644, @array;
```

不过chmod的定义只是：

```
chmod LIST
```

所以你可以写为：

```
unshift @array, 0644;  
chmod @array;
```

如果列表的第一个参数不是一个合法的模式，chmod会失败，不过这是一个运行时语义问题，与这个调用的语法无关。如果语义要求首先传递一些特殊的参数，在正文描述中会指出这些限制。

与简单的LIST函数不同，其他函数还会施加额外的语法限制。例如，push的语法概要如下所示：

```
push ARRAY, LIST
```

这表示push需要一个适当的数组作为它的第一个参数，不过并不关心其余的参数。这正是末尾LIST的含义（LIST总是放在最后，这样可以涵盖所有其余的值）。如果语法概要中在LIST前面还包含其他参数，这些参数会由编译器做语法区分，而不只是在后来运行时由解释器做语义区分。这些参数不会在列表上下文中计算。它们可能在标量上下文计算，或者可能是特殊的引用参数，如push中的数组（在后面的语法描述中会告诉你究竟是什么类型的参数）。

有些操作直接基于C的库函数，对于这些操作，我们不打算原样照搬你的系统的文档。如果函数描述中提到请参见function(2)，这说明你要查看该函数相应的C版本，更多地了解它的语义。小括号中的数指示了可以在系统程序员手册中的哪一节找到这个手册页（假设你已经安装了手册页）。当然，如果你还没有安装，说明在那一小节找不到这个手册页。

这些手册页可能会介绍依赖于具体系统的行为，如shadow口令文件、访问控制列表等等。有些Perl函数来自于Unix的C库函数，在非Unix平台上尽管不能直接使用这些函数调用，但也有了相应的模拟函数。例如，尽管你的操作系统可能不支持flock(2)或fork(2)系统调用，但Perl可能会尽其所能使用你的平台提供的原生功能来模拟这些函数。



有时，你会发现有文档的C函数比相应Perl函数的参数多。一般地，所缺少的参数往往是Perl已经知道的东西，如前一个参数的长度，所以无需在Perl中提供。其余的差异则是由于Perl和C以不同的方式指定文件句柄和成功/失败值所导致的。

Perl中有些函数是同名的系统调用的包装器（如`chown(2)`, `fork(2)`, `closedir(2)`等），一般地，这些函数成功时都返回`true`，否则返回`undef`，在后面的描述中会提到。这与C库中相应操作的接口不同，它们在失败时都返回-1。也有几个例外：`wait`、`waitpid`和`syscall`。另外系统调用还会在失败时设置特定的`$_($OS_ERROR)`变量。而其他函数不会这样，除非意外设置。

对于既可以在标量上下文又可以在列表上下文中使用的函数，在标量上下文中通常返回一个`false`值（一般是`undef`）来指示失败，而在列表上下文中会返回`null`列表来指示失败。如果成功执行，通常会返回一个（在当前上下文）计算为`true`的值来指示成功。

要记住以下规则：一个函数在列表上下文中的行为与其在标量上下文中的行为没有关联，反之亦然。它们可能做两种完全不同的工作。

每个函数都知道它在哪个上下文中调用。一个函数在列表上下文中调用时，会返回一个列表，同样的这个函数在标量上下文中调用时，会返回一个最合适的值。有些函数会返回在列表上下文中可能返回的列表的长度。有些操作符将返回列表中的第一个值。有些函数则返回列表中的最后一个值。如果可以按数字或名字查找某个值，有些函数甚至会返回“其他”值。还有一些函数会返回成功操作的数目。一般来讲，Perl函数会按你希望的去做，除非你希望保持一致。

最后再提醒一点：使用“字节”和“字符”这两个词时我们力求一致。由于历史原因，这两个词彼此是混淆的（它们自己也很含糊）。不过，我们说到“字节”时，是指一个序数值为8位的字符。而说到“字符”时，通常是指一个抽象Unicode码点。C程序员习惯于用`char`变量表示一个字符，直到后来`char`已经无法提供足够的支持。如今，`int`成为新的`char`。码点是程序员可见的字符，这是对应单个Unicode实体的非负整数，有时正式的名字就是字符。

目前，除了Perl正则表达式库之外，很少有Perl函数处理字形，不过字形是在码点之上的下一个抽象层。字形是用户可见的字符，可能由多个程序员可见的字符组成。CR+LF就是字形的一个例子，它占有两个码点。另一个很好的例子是`ō`，它可能占1到3个码点，这取决于规范化方式：采用NFC是“`\x{22D}`”，采用NFD是“`\x{6F}\x{303}\x{304}`”，或者未采用这两种规范化方式时则为“`\x{F5}\x{304}`”。这一章中，如果我们谈到字符，这是指码点，如果谈到字节，则只是表示小于256的未解码的序数值。



# 按类别组织的Perl函数

以下是按类别组织的Perl函数以及类函数关键字。有些函数可能出现在多个类别下面。

## 标量处理

chomp, chop, chr, crypt, fc, hex, index, lc, lcfirst, length, oct, ord, pack, q//, qq//, reverse, rindex, sprintf, substr, tr///, uc, ucfirst, y///

## 正则表达式和模式匹配

m//, pos, qr//, quotemeta, s///, split, study

## 数值函数

abs, atan2, cos, exp, hex, int, log, oct, rand, sin, sqrt, srand

## 数组处理

pop, push, shift, splice, unshift

在v5.12中，如果确实需要，还可以对数组使用each、keys和values。

## 列表处理

grep, join, map, qw//, reverse, sort, unpack

## 散列处理

delete, each, exists, keys, values

## 输入和输出

binmode, close, closedir, dbmclose, dbmopen, die, eof, fileno, flock, format, getc, print, printf, read, readdir, readpipe, rewinddir, say, seek, seekdir, select (准备好的文件描述符), syscall, sysread, sysseek, syswrite, tell, telldir, truncate, warn, write

## 定长数据和记录

pack, read, syscall, sysread, sysseek, syswrite, unpack, vec

## 文件句柄、文件和目录

-X, chdir, chmod, chown, chroot, fcntl, glob, ioctl, link, lstat, mkdir, open, opendir, readlink, rename, rmdir, select (准备好的文件描述符), select (输出文件句柄), stat, symlink, sysopen, umask, unlink, utime

## 程序控制流

caller, continue, die, do, dump, eval, exit, \_\_FILE\_\_, goto, last, \_\_LINE\_\_, next, \_\_PACKAGE\_\_, redo, return, sub, wantarray

## 作用域

caller, import, local, my, no, our, package, state, use

state只有当启用“state”特性或者前面有CORE::前缀时才可用。参见feature。或者可以在当前作用域包含use v5.10或以后版本。

## switch特性

break, continue, default, given, when

除非continue作为一个表达式而不是一个块，否则只有启用了“switch”特性时这些函数才可用。或者可以在当前作用域包含use v5.10或以后版本。参见第4章中“given语句”一节。

## 其他

defined, dump, eval, formline, lock, prototype, reset, scalar, undef, wantarray

## 进程和进程组

alarm, exec, fork, getpgrp, getppid, getpriority, kill, pipe, qx//, setpgrp, setpriority, sleep, system, times, wait, waitpid

## 库模块

do, import, no, package, require, use

## 类和对象

bless, dbmclose, dbmopen, package, ref, tie, tied, untie, use

## 底层套接字访问

accept, bind, connect, getpeername, getsockname, getsockopt, listen, recv, send, setsockopt, shutdown, socket, socketpair

## 系统 V进程间通信

msgctl, msgget, msgrcv, msgsnd, semctl, semget, semop, shmctl, shmget, shmread, shmwrite

## 获取用户和组信息

endgrent, endhostent, endnetent, endpwent, getgrent, getgrgid, getgrnam, getlogin, getpwent, getpwnam, getpwuid, setgrent, setpwent

## 获取网络信息

endprotoent, endservent, gethostbyaddr, gethostbyname, gethostent, getnetbyaddr, getnetbyname, getnetent, getprotobyname, getprotobynumber, getprotoent, getservbyname, getservbyport, getservent, sethostent, setnetent, setprotoent, setservent

时间

gmtime, localtime, time, times

与Unicode相关的函数

binmode, chomp, chop, chr, dbmopen, fc, getc, index, lc, lcfirst, length, m//, my, open, ord, our, pack, package, pos, print, printf, quotemeta, read, readline, reverse, rindex, s///, seek, sort, split, sprintf, state, substr, sysopen, sysread, sysseek, syswrite, tell, tr///, truncate, uc, ucfirst, unpack, write, y///

## 按字母顺序组织Perl函数

以下很多函数名都带有注解。下面给出这些注解的含义：

- **\$\_** 使用\$\_ (\$ARG)作为默认变量。
- **\$\_!** 对系统调用错误设置\$\_!(\$OS\_ERROR)。
- **\$\_@** 产生异常。使用eval来捕获\$\_@ (\$EVAL\_ERROR)。
- **\$\_?** 子进程退出时设置\$\_? (\$CHILD\_ERROR)。
- **T** 污染返回的数据。
- **T** 某些系统、本地化环境或句柄设置下污染返回的数据。
- **X ARG** 如果给出一个类型不合适的参数，则产生一个异常。
- **X RO** 如果修改一个只读的目标，则产生一个异常。
- **X T** 如果输入污染的数据，则产生一个异常。
- **X U** 如果当前平台上未实现，则产生一个异常。
- **X WIDE** 如果传入的字符串中包含序数值大于255的字符，则产生一个异常。

如果输入污染的数据，有些函数会返回污染的数据，不过这些函数没有特别标出，因为大多数函数都是如此。具体来讲，如果在%ENV或@ARGV上使用任何函数，都会得到污染的数据。

标有**X ARG**的函数可能需要某个特定类型的参数，如果没有接收到满足要求的参数，就会产生一个异常（如I/O操作需要的文件句柄，使用bless祝福时需要的引用等）。

标有**X RO**的函数有时需要修改其参数。如果不能修改参数（因为参数标志为只读），则会产生一个异常。只读变量有很多例子，如包含模式匹配所捕获数据的特殊变量，以及实际上是常量别名的变量。



标有 **X<sub>U</sub>** 的函数可能并没有在所有平台上都实现。尽管其中一些是按Unix C库中的函数来命名，但不要认为你运行的不是Unix就不能调用这些函数。很多函数已经在其他平台上有了模拟函数，甚至是你从未想过的，如Win32系统上的fork。关于可移植性和系统特定函数的行为，更多信息请参见*perlport*手册页，另外还可以参考随Perl移植版本发布的与特定平台相关的文档。

如果为标有 **X<sub>WIDE</sub>** 的函数传入一个未解码的字符串（其中有些字符过大，无法放在一个字节值中），则会产生一个异常。

可能产生其他异常的函数会用 **\$@** 标志，包括抛出越界错误的数学函数，如sqrt(-1)。

## abs

**\$<sub>-</sub>**

```
abs VALUE
abs
```

这个函数返回其参数的绝对值。

```
$diff = abs($first - $second);
```

在这里以及后面的例子中，好的编码风格（和strict pragma）要求增加一个my修饰符来声明一个新的词法作用域变量，如下所示：

```
my $diff = abs($first - $second);
```

不过，为简洁起见，大多数例子中都省略了my。你要假设所有这种变量已经在前面声明过。

## accept

**\$!** **X<sub>ARG</sub>** **X<sub>U</sub>**

```
accept SOCKET, PROTOCKET
```

这个函数由服务器进程使用，这些进程希望监听来自客户端的套接字连接。*PROTOCKET* 必须是一个已经通过socket操作符打开而且绑定到某个服务器网络地址或绑定到INADDR\_ANY的文件句柄。调用这个函数时，执行会挂起，直到建立连接。建立连接时，SOCKET文件句柄会打开，并关联到新建立的连接。原来的*PROTOCKET*保持不变，它唯一的作用就是要克隆为一个真正的套接字。如果调用成功，这个函数返回所连接的地址，否则返回false。

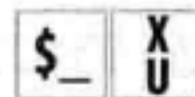
例如：

```
unless ($peer = accept(SOCKET, PROTOCKET)) {
    die "Can't accept a connection: $!";
}
```

在支持“执行时关闭”（close-on-exec）标志的系统上，会为新打开的文件描述符设置这个标志，由`$^F`（`$SYSTEM_FD_MAX`）的值确定。

参见`accept(2)`。还可以参见第15章中“套接字”一节的例子。

## alarm



```
alarm EXPR
alarm
```

这个函数告诉操作系统在过去`EXPR`秒（墙上时钟时间）之后向当前进程发送一个`SIGALRM`信号。

一次只能有一个定时器起作用。每个调用会禁用前一个定时器，`EXPR`为0时会取消前一个定时器，而且不启动新的定时器。返回值是前一个定时器剩余的时间。

```
print "Answer me within one minute, or die: ";
alarm(60);           # 1分钟后关闭程序
$answer = <STDIN>;
$timeleft = alarm(0); # 清除警报
say "You had $timeleft seconds remaining";
```

混合使用`alarm`和`sleep`调用通常是一个错误，因为很多系统都使用`alarm(2)`系统调用机制来实现`sleep(3)`。以往，由于计秒的方式不同，耗用时间可能与你指定的时间差多达1秒。另外，如果系统很忙，可能不能立即运行你的进程。参见第15章，了解有关信息处理的内容，如如何使用警报让慢操作超时。

要实现粒度更细的警报（小于1秒），`Time::HiRes`模块提供了相应的函数。还有一个更巧妙的方法，可以使用4参数版本的`select`（前3个参数保持为未定义），或者使用Perl的`syscall`函数来访问`setitimer(2)`（如果你的系统提供支持）。

## atan2

```
atan2 Y, X
```

这个函数返回`Y/X`反正切值，范围在 $-\pi$ 到 $+\pi$ 之间。得到 $\pi$ 近似值的一种快捷方法是：

```
$pi = atan2(1,1) * 4;
```

对于正切操作，可以使用`Math::Trig`或`POSIX`模块的`tan`函数，或者直接使用我们熟悉的关系来实现：

```
sub tan { sin($_[0]) / cos($_[0]) }
```

如果其中一个参数为0或两个参数都为0，返回值由具体实现定义，有关的更多信息参见你的`atan2(3)`手册页。

## bind

\$!	X ARG	X T	X U
-----	----------	--------	--------

`bind SOCKET, NAME`

这个函数为一个未命名但是已经打开的套接字（由`SOCKET`文件句柄指定）命名，使得其他进程可以找到这个套接字。如果成功，这个函数返回`true`，否则返回`false`。`NAME`应当是有适当类型的套接字打包地址。

```
use Socket;
$port_number = 80;      # 假装我们作为一个Web服务器
$sockaddr = sockaddr_in($port_number, INADDR_ANY);
bind(SOCK, $sockaddr) || die "Can't bind $port_number: $!";
```

参见`bind(2)`。另外参见第15章“套接字”一节中的例子。正常情况下，应当使用标准`IO::Socket`模块提供的更高层套接字接口。

## binmode

X ARG
----------

`binmode FILEHANDLE, IOLAYER`  
`binmode FILEHANDLE`

这个函数使`FILEHANDLE`有`IOLAYER`参数指定的语义。如果省略`IOLAYER`，则对这个文件句柄应用二进制（或“原始”）语义。如果`FILEHANDLE`是一个表达式，则根据情况将它的值用作文件句柄的名字或者作为文件句柄引用。如果成功，这个函数返回`true`，否则返回`false`。

`binmode`函数应当在`open`之后调用，但是需要在对文件句柄完成任何I/O之前调用。要对一个文件句柄重置模式，唯一的办法就是重新打开这个文件，因为各个不同层可能已经在不同缓冲区中积累了各种数据。这个限制将来可能会放宽。

有些操作系统的运行时库会区别文本文件和二进制文件，原先`binmode`就主要用于这样一些操作系统。在这些系统上，`binmode`的作用是关闭默认的文本语义。不过，随着Unicode和众多不同存储编码的到来，所有系统上的程序都必须对这种区别有所认识。

如今只有一种二进制文件（就Perl而言），不过有很多种文本文件，Perl希望采用同一种方式处理这些不同的文本文件。所以Perl对于Unicode文本有一个内部格式：UTF-8。

由于有多种文本文件，输入的文本文件通常需要转换为UTF-8，然后在输出时再转换回某种遗留字符集或者Unicode的其他表示。

可以使用I/O层告诉Perl具体（或者不那么具体）如何完成这些转换。例如，“`:text`”会告诉Perl完成通用的文本处理，而不告诉Perl要做哪种类型的文本处理。

不过像“`:utf8`”和“`:encoding(Latin1)`”等I/O层会告诉Perl读写哪种文本格式。



另一方面，“:raw” I/O层会告诉Perl不要对数据做任何额外处理。

对于I/O层如何工作，更多内容参见open函数。后面将介绍没有IOLAYER参数时binmode如何工作；即binmode的历史含义，这就等价于：

```
binmode FILEHANDLE, ":raw";
```

除非另行说明，否则Perl会假设新打开的文件应当采用文本模式读或写。文本模式意味着\n（换行符）将是内部的行终止符。所有系统都使用\n作为内部行终止符，但是它具体表示什么会随不同的系统、不同的设备，甚至不同的文件而有所不同，这取决于你如何访问这个文件。在遗留系统中（包括MS-DOS和VMS），你的程序看到的\n可能并不是磁盘上真正存储的东西。例如，操作系统存储文本文件时可能会有\cM\cJ序列，它会在输入时完成转换，使得在程序中会显示为\n。在从程序输出时，再把\n转换回\cM\cJ。binmode函数禁用了这些系统上的这种自动转换。

如果没有IOLAYER参数，binmode在Unix下（包括Mac OS X）没有任何作用，它们都使用\n来结束各行，并把它表示为单个字符（不过，这可能是一个不同的字符：Unix使用\cJ，而Unix前的Macs使用cM。这并不重要）。

下面的例子显示了一个Perl脚本如何从文件读取一个GIF图像，并把它打印到标准输出上。有些系统会改变直接量数据，使它不同于数据的实际物理表示，在这些系统上，必须准备两个句柄。尽管可以在打开GIF时直接使用一个“:raw”，但是对于预打开的文件句柄（如STDOUT），这一点并不容易做到：

```
binmode(STDOUT, ":raw")
|| die "couldn't binmode STDOUT to raw: $!";
open(GIF, "< :raw", "vim-power.gif")
|| die "Can't open vim-power.gif: $!";
while (read(GIF, $buf, 1024)) { # 现在是字节，不是字符
    print STDOUT $buf;
}
```

需要说明，如果使用内置UTF-8层，如下所示：

```
binmode(HANDLE, ":utf8");
```

如果它是一个输入句柄，你必须做好准备自行处理编码错误，因为在v5.14中，对于不合法的UTF-8，默认情况下态度会过于宽松。要处理编码错误，最快速同时可能也是最好的办法是根本不允许这些错误出现。

```
use warnings FATAL => "utf8";
```

即使你使用类似下面的代码实现了Encode模块：

```
binmode(HANDLE, ":encoding(utf8)")
```

还应当把UTF-8警告变为致命错误，如上所示，因为如果不这样做，即使有错误你也不会得到异常（要相信，宁愿得到异常也不要乱七八糟的文本。不过这些都不是好事）。

## bless



```
bless REF, CLASSNAME
bless REF
```

这个函数告诉指示对象（由引用`REF`指示）它现在是`CLASSNAME`包中的一个对象，或者如果没有指定`CLASSNAME`，则是当前包中的一个对象。如果`REF`不是一个合法的引用，会产生一个异常。为方便起见，`bless`会返回这个引用，因为这通常是构造函数子例程中的最后一个函数。例如：

```
$pet = Beast->new(TYPE => "cougar", NAME => "Clyde");

# 然后在Beast.pm中：
sub new {
    my $class = shift;
    my %attrs = @_ ;
    my $self = { %attrs };
    return bless($self, $class);
}
```

要把对象祝福到某个类，这个类名`CLASSNAME`应当混合大小写。包含全小写名的名字空间是为Perl pragmata（编译器指令）内部使用而保留的。内置类型（如“SCALAR”，“ARRAY”，“HASH”，“UNIVERSAL”等等）都是大写名，所以也要避免使用这些包名。

要确保`CLASSNAME`不是false；并不支持祝福到false包，而且这可能导致不可预测的行为。

Perl中有`bless`（祝福）而没有相应的`curse`（诅咒）操作符，这并不是一个bug（不过确实有一个`sin`操作符）。有关对象祝福的更多信息请参见第12章。

## break

```
break
```

比正常（在`when`子句结束前）提前从`given`块退出。这个关键字由`switch`特性启用，更多信息请参见第29章中的`feature pragma`。

## caller

```
caller EXPR
caller
```

这个函数返回当前子例程调用等方面的堆栈信息。如果没有参数，它会返回包名、文件名和调用当前执行的子例程的相应行号：

```
($package, $filename, $line) = caller;
```

下面是一个极为挑剔的函数的例子，这里利用了第2章介绍的特殊token `__PACKAGE__` 和 `__FILE__`：

```
sub careful {
    my ($package, $filename) = caller;
    unless ($package eq __PACKAGE__ && $filename eq __FILE__) {
        die "You weren't supposed to call me, $package!";
    }
    say "called me safely";
}

sub safecall {
    careful();
}
```

调用时如果有参数，`caller`会计算`EXPR`，作为从当前栈帧回退的栈帧数。例如，参数为0表示当前栈帧，1表示调用者，2表示调用者的调用者，依此类推。这个函数还能报告额外的信息，如下所示：

```
my $i = 0;
while (my ($package, $filename, $line, $subroutine,
          $hasargs, $wantarray, $evaltext, $is_require,
          $hints, $bitmask, $hinthash) = caller($i++))
{
    ...
}
```

如果这个栈帧是一个子例程调用，它有自己的`@_`数组（而不是从它的调用者借来的），那么`$hasargs`为true。否则，如果栈帧不是一个子例程调用，而是一个`eval`，`$subroutine`可能为“(eval)”。如果是这样，会设置额外的元素`$evaltext`和`$is_require`：如果栈帧由一个`require`或`use`语句设置，而且`$evaltext`包含`eval EXPR`语句的文本，那么`$is_require`为true。特别是对于`eval BLOCK`语句，`$filename`为“(eval)”，但是`$evaltext`未定义（还要注意，每个`use`语句会在一个`eval EXPR`帧中创建一个`require`帧）。`$hints`、`$bitmask`和`$hinthash`都是内部值，请不要理会这些变量，除非你是实现Perl的核心成员<sup>注2</sup>。

还有更深一层的魔法，`caller`还会把数组`@DB::args`设置为传入给定栈帧中的参数——不过只有从DB包调用时才会这样做。参见第18章。

当心在`caller`有机会得到信息之前，优化器可能已经优化了调用帧。这意味着`caller(N)`可能不会返回你期望的 $N > 1$ 调用帧的有关信息。具体地，`@DB::args`可能包含前一次调用`caller`的信息。

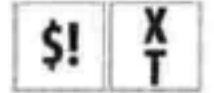
---

注2： `$hinthash`是一个散列引用，这个散列中包含编译调用者时的`%^H`的值，或者如果`%^H`为空则为`undef`。不要修改这个散列的值，因为它们是存储在操作树（optree）中的实际值。



另外还要知道，设置@DB::args是尽力而为（best effort），只是为了调试或生成回溯记录，不要依赖这种方法。特别地，@\_包含调用者的当前@\_数组的别名。Perl并未在子例程入口取@\_的一个快照，所以@DB::args会反映继其调用之后子例程对@\_的所有修改。另外，@DB::args（如@\_）不包含其元素的显式引用，所以在某些情况下，其元素可能已经释放或者已经重新分配给其他变量或临时值。最后一点，当前这个实现有一个副作用：只能取消shift @\_的作用（而pop或splice不能取消），如果取@\_的一个引用，你可能会失败。所以@DB::args实际上是@\_当前状态和初始状态的混合产物。务必要当心。

## chdir



```
chdir EXPR
chdir
```

这个函数将当前进程的工作目录改为*EXPR*（如果可能）。省略*EXPR*时，如果设置了\$ENV{HOME}则使用\$ENV{HOME}，否则使用\$ENV{LOGDIR}，这些往往是进程的主目录。成功时这个函数返回true，否则返回false。

```
chdir("$prefix/lib") || die "Can't cd to $prefix/lib: $!";
```

参见Cwd模块，它允许你自动跟踪你的当前目录。

在支持fchdir(2)的系统上，可以传入一个文件句柄或目录句柄作为*EXPR*。在不支持fchdir(2)的系统上，传递句柄会产生一个运行时异常。

## chmod



```
chmod LIST
```

这个函数改变文件列表的权限。列表中的第一个元素必须是一个表示模式的数字，与chmod(2)系统调用中类似。这个函数返回成功改变的文件数。例如：

```
$cnt = chmod 0755, "file1", "file2";
```

\$cnt会设置为0、1或2，这取决于改变了多少个文件。成功要通过没有错误来体现，而不是通过具体的改变来体现，因为文件可能与操作前的模式相同。出现错误可能意味着你缺少足够的权限来改变文件模式，因为你不是文件的所有者，也不是超级用户。检查\$!来查看失败的具体原因。

下面是一个更典型的使用：

```
chmod(0755, @executables) == @executables
|| die "couldn't chmod some of @executables: $!";
```

如果需要知道哪些文件不允许这种改变模式，可以使用类似下面的代码：

```
@cannot = grep {not chmod(0755, $_) } "file1", "file2", "file3";
die "$0: could not chmod @cannot" if @cannot;
```

这个惯用法使用了`grep`函数，从而只选择列表中使`chmod`函数失败的那些元素。

在支持`fchmod(2)`的系统上，可以在参数列表传入一个文件句柄。在不支持`fchmod(2)`的系统上，传递句柄会产生一个运行时异常。为了便于识别，文件句柄必须作为一个类型团或类型团引用传递；字符串会被认为是文件名。

使用非直接量模式数据时，可能需要使用`oct`函数把一个八进制串转换为一个数。这是因为Perl不会因为字符串中有一个前导“0”就自动假设这个字符串包含八进制数。

```
$DEF_MODE = 0644;          # 这里不能使用引号!
PROMPT: {
    print "New mode? ";
    $strmode = <STDIN>;
    exit unless defined $strmode;      # 检查eof
    if ($strmode =~ /\s$/) {           # 检查空行
        $mode = $DEF_MODE;
    }
    elsif ($strmode !~ /\d+$/) {
        say "Want numeric mode, not $strmode";
        redo PROMPT;
    }
    else {
        $mode = oct($strmode);         # 将"755"转换为0755
    }
    chmod $mode, @files;
}
```

这个函数对数值模式的处理与Unix `chmod(2)`系统调用类似。如果想要一个类似于`chmod(1)`命令提供的符号接口，参见CPAN上的File::chmod模块。

还可以从Fcntl模块导入符号常量`S_I*`：

```
use Fcntl ":mode";
chmod S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH, @executables;
```

有些人认为这比0755更可读。可以试试看。

## chomp



```
chomp VARIABLE
chomp LIST
chomp
```

这个函数（正常情况下）从变量中包含的一个字符串中删除末尾的换行符。这是`chop`（见后面的介绍）的一个更安全的版本，这个函数对不是以换行符结尾的字符串没有影响。更确切地讲，它会删除终止符字符串（对应`$/`的当前值），而不只是最后一个字符。

不同于chop，chomp会返回所删除的字符个数。如果\$/是""（段落模式），chomp会从所选择的字符串（或者如果对一个LIST调用chomp，则可能是多个字符串）删除所有末尾的换行符。对于slurp模式（\$/ = undef）或定长记录模式（\$/是一个整数的引用），chomp什么也不做。不能对一个直接量调用chomp，这个函数只能应用于变量。对散列调用chomp只会处理其中的值，而不处理键。

例如：

```
while (<PASSWD>) {  
    chomp; # 避免最后一个字段出现\n  
    @array = split /:/;  
    ...  
}
```

I/O层可以覆盖\$/变量的值，并标志字符串应当如何chomp。这有一个好处，I/O层可以识别多种行终止符（如Unicode段落和行分隔符），而且仍能安全地执行chomp，而不论当前行的终止符是什么。

chomp函数目前还不够聪明，还不能处理Unicode换行序列，其正则表达式元字符为\R。你可以自己来处理：

```
s/\R/\n/g; # 将所有Unicode换行符转换为\n
```

或者，有时可以这样做：

```
my @paras = split /\R+/, our $file_contents;
```

不过，如果你想保留这个换行序列，最好这样做：

```
our $line =~ s/(\R?)\z//;  
my $terminator = $1;
```

## chop



```
chop VARIABLE  
chop LIST  
chop
```

这个函数删除一个字符串变量的最后一个字符，并返回所删除的字符。chop函数主要用于从一个输入记录的末尾删除换行符，它比使用替换效率要高。如果你要做的只是这些，那么更安全的做法是使用chomp，因为不论字符串是什么chop都会缩短字符串，而chomp更有选择性。

不能对一个直接量调用chop，只能应用于变量。如果对一个变量LIST调用chop，会处理这个列表中的各个串：

```
@lines = `cat myfile`;
```



```
chop @lines;
```

可以对任何作为左值的变量调用chop，包括赋值：

```
chop($cwd = `pwd`);  
chop($answer = <STDIN>);
```

这与下面不同：

```
$answer = chop($tmp = <STDIN>);      # 不正确
```

这会在\$answer中加入一个换行符，因为chop会返回所删除的字符，而不是其余的字符串（其余的字符串保存在\$tmp中）。要得到我们想要的结果，一种方法是使用substr：

```
$answer = substr <STDIN>, 0, -1;
```

不过更常用的方法是：

```
chop($answer = <STDIN>);
```

一般情况下，chop可以使用substr表示：

```
$last_char = chop($var);  
$last_char = substr($var, -1, 1, ""); # 作用相同
```

一旦了解这种等价性，可以使用它来完成更大的chop删除。要删除多个字符，可以使用substr作为左值，为它赋一个null串。下面会删除\$caravan的最后5个字符：

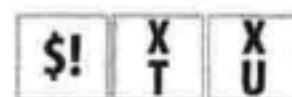
```
substr($caravan, -5) = "";
```

负下标会让substr从字符串末尾倒数，而不是从前面向后数。要保存所删除的字符，可以使用4参数形式的substr，5个字符一组地进行删除：

```
$tail = substr($caravan, -5, 5, "");
```

处理码点而不是字形是一件非常危险的事情。Perl并没有一个字形模式，所以你必须自己来处理。考虑一个类似*naïveté*的单词，这在NTD中实际上是nai\x{308}vete\x{301}。如果使用chop，最后不会得到*naïvet*，而会得到*naïvete*。需要使用s/\X\z//来删除一个字形而不是码点。CPAN Unicode::GCString模块对此很有帮助。

## chown



chown LIST

这个函数改变一组文件的所有者和所属的组。列表中前两个元素必须是数值UID和GID（必须是这个顺序）。任何一个值为-1时，大多数系统都会把它解释为保留这个值不变。函数会返回成功改变的文件的个数。例如：

```
chown($uidnum, $gidnum, "file1", "file2") == 2
|| die "can't chown file1 || file2: $!";
```

`$cnt`将设置为0、1或2，这取决于改变了多少个文件（这是从操作成功的角度来讲，而不是看操作之后所有者是否不同）。

下面是一个更典型的用法：

```
chown($uidnum, $gidnum, @filenames) == @filenames
|| die "can't chown @filenames: $!";
```

下面的子例程接收一个用户名，查找它的用户ID和组ID，并完成`chown`：

```
sub chown_by_name {
    my($user, @files) = @_;
    chown((getpwnam($user))[2,3], @files) == @files
        || die "can't chown @files: $!";
}

chown_by_name("fred", glob("*.c"));
```

不过，你可能不希望像前一个函数那样改变组，因为`/etc/passwd`文件会把各个用户与一个组关联，即使从`/etc/group`来看这个用户可能是多个从属组的成员。一种候选方法是为GID传入一个-1，这样不会改变文件的组。如果传入-1作为UID，同时传递一个合法的GID，可以建立组而不改变所有者。

在支持`fchown(2)`的系统上，还可以在参数表中传递文件句柄。在不支持`fchown(2)`的系统上，传递文件句柄会产生一个运行时异常。为了便于识别，文件句柄必须作为一个类型团或类型团引用传递：字符串会被认为是文件名。

在大多数系统上，不允许改变文件的所有者，除非你是超级用户，不过应该可以将组改为任何从属组。在不安全的系统上，这些限制可能会放松，不过这不是一个可移植的假设。在POSIX系统上，可以如下检查应用哪个规则：

```
use POSIX qw(sysconf _PC_CHOWN_RESTRICTED);
# 只有作为超级用户或者在一个有权限的系统上才可以尝试
if ($? == 0 || !sysconf(_PC_CHOWN_RESTRICTED) ) {
    chown($uidnum, -1, $filename)
        || die "can't chown $filename to $uidnum: $!";
}
```

## chr

\$\_

```
chr NUMBER
chr
```

这个函数返回Unicode字符集中`NUMBER`（截断为一个整数）表示的字符。例如，`chr(65)`是“A”，LATIN SMALL LETTER A，`chr(0x2122)`是“™”，TRADE MARK SIGN。如

果要得到与chr相反的功能，可以使用ord。

如果NUMBER是负数，这个函数会生成Unicode REPLACEMENT字符，U+FFFD<sup>注3</sup>。

需要说明，由于向后兼容性方面的原因，码点在128到255之间的字符在内部默认地并不采用UTF-8编码。你不用注意这一点，不过如果确实注意到了，要知道这是为了保证向后兼容性。

如果要按名指定字符而不是按数字指定（例如，指定“\N{WHITE SMILING FACE}”表示Unicode微笑符号“☺”），可以参见第29章“字符名”一节。要把一个字符编号转换为它的正式名字而不是字符本身，参见这个pragma的charnames::viacode函数。

## chroot

\$_	\$!	X	X
		T	U

```
chroot FILENAME
chroot
```

如果成功，FILENAME会变成当前进程的新的根目录，即以“/”开头的路径名的起点。这个目录通过exec调用继承，chroot调用之后用fork派生的所有子进程都会继承这个目录。chroot没有办法撤销。出于安全方面的原因，只有超级用户可以使用这个函数。

下面的代码与很多FTP服务器所做的工作大致相同：

```
chroot((getpwnam("ftp"))[7])
|| die "can't do anonymous ftp: $!";
```

这个函数在非Unix系统上可能不能使用。参见chroot(2)。

## close

\$_	\$?	X
		ARG

```
close FILEHANDLE
close
```

这个函数在刷新输出所有IO缓冲区之后会关闭与FILEHANDLE关联的文件、套接字或管道。如果省略参数，它会关闭当前选择的文件句柄。如果关闭成功，则返回true，否则返回false。如果接下来马上要对FILEHANDLE完成另一个open，那么不用关闭这个文件句柄，因为下一个open会为你关闭文件句柄，不过这样一来你可能会漏掉一些可能出现的错误（参见open）。不过，对一个输入文件显式调用close会重置行计数器（\$.）；由open隐式完成文件关闭则不会重置这个计数器。

FILEHANDLE可能是一个表达式，它的值可以用作为一个间接文件名（可以是真实的文件句柄名，或者是任何文件句柄对象的引用）。

---

注3： 除非有bytes pragma，在这种情况下，会使用值的低8位。



文件句柄来自一个管道式open时，如果底层系统调用失败，或者管道另一端的程序以非0状态退出，close会返回false。对于后一种情况，close强制\$!(\$OS\_ERROR)为0。所以，如果一个管道上的close返回一个非0状态，要检查\$!来确定问题是否出自管道本身（非0值），还是出自另一端的程序（0值）。不论哪一种情况，\$?(\$CHILD\_ERROR)和\${^CHILD\_ERROR\_NATIVE}都包含与管道另一端相关联的命令的等待状态值（参见system下的解释）。例如：

```
open(OUTPUT, "| sort -rn | lpr -p") # 完成sort和lpr的管道
|| die "Can't start sortlpr pipe: $!";
print OUTPUT @lines;                # 把内容打印到输出
close(OUTPUT)                        # 等待sort完成
|| warn $!? "Syserr closing sortlpr pipe: $!"
    : "Wait status $? from sortlpr pipe";
```

用dup(2)复制一个管道所生成的文件句柄会当作一个普通的文件句柄，所以在这个文件句柄上close不会等待子进程。必须关闭原来的文件句柄才会等待子进程。例如：

```
open(NETSTAT, "netstat -rn |")
|| die "can't run netstat: $!";
open(STDIN, "<&NETSTAT")
|| die "can't dup to stdin: $!";
```

如果关闭上面的STDIN，就没有等待；如果关闭NETSTAT，则有等待。

如果想要以某种方式自行获得一个退出的管道子进程，close会失败。假如你有你自己的一个\$SIG{CHLD}处理器，会在管道子进程退出时触发，或者如果你有意在open调用返回的进程ID上调用waitpid，就会发生这种情况。

## closedir

\$!	X	X
ARG	ARG	U

closedir DIRHANDLE

这个函数关闭opendir打开的一个目录，并返回这个操作是否成功。参见readdir下的例子。DIRHANDLE可以是一个表达式，其值可以用作为一个间接目录句柄，通常是真实的目录句柄名或自动生成的句柄对象。

## connect

\$!	X	X	X
ARG	ARG	T	U

connect SOCKET, NAME

这个函数初始化与另一个进程的连接，该进程在等待一个accept。如果成功，这个函数返回true，否则返回false。NAME应当是有适当类型的套接字打包网络地址。例如，假设SOCK是之前创建的一个套接字：

```
use Socket;
```

```
my ($remote, $port) = ("www.perl.com", 80);
my $destaddr = sockaddr_in($port, inet_aton($remote));
connect(SOCK, $destaddr)
    || die "Can't connect to $remote at port $port: $!";
```

要断开与一个套接字的连接，可以使用`close`或`shutdown`。参见第15章“套接字”一节中的例子。另外可以参见`connect(2)`。要完成大多数套接字操作，更好的做法是使用标准`IO::Socket`模块提供的更高层的接口。

## continue

这通常是一个流控制语句而不是一个函数。如果有一个`continue`关联到一个`BLOCK`（通常在一个`while`或`foreach`中），它通常在将要再次计算条件之前执行，类似于`for(;;)`循环的第三部分。因此，可以用来让一个循环变量自增，尽管已经通过`next`语句继续循环（这与C的`continue`语句类似）。

`last`、`next`或`redo`可以出现在`continue`块中；`last`和`redo`表现得好像它们已经在主代码块中执行。`next`也是这样，不过由于它会执行一个`continue`块，可能更有意思。

```
while (EXPR) {
    ### redo总放在这里
    do_something;
} continue {
    ### next总放在这里
    do_something_else;
    # 然后回到最上面重新检查EXPR
}
### last总放在这里
```

省略`continue`部分等价于使用一个空的`continue`，从逻辑上讲是可以的，所以`next`直接回到最上面检查循环顶部的条件。参见第4章中“循环控制”一节。

不过，如果启用了“`switch`”特性，`continue`也是一个操作符，会退出当前`when`或`default`块，而且默认地会继续执行到下一个分支。另外参见第4章中“`given`语句”一节。

## COS



```
cos EXPR
cos
```

这个函数返回`EXPR`（用弧度表示）的余弦。例如，下面的脚本会打印用度表示的角度的一个余弦表：

```
# 这是得到度到弧度转换的一种懒办法

$pi = atan2(1,1) * 4;
```

```

$pi_over_180 = $pi/180;

# 打印表格
for ($deg = 0; $deg <= 90; $deg++) {
    printf "%3d %7.5f\n", $deg, cos($deg * $pi_over_180);
}

```

对于反余弦操作，可以使用`Math::Trig`或`POSIX`模块的`acos`函数，或者使用以下关系：

```
sub acos { atan2( sqrt(1 - $_[0] * $_[0]), $_[0] ) }
```

## crypt



```
crypt PLAINTEXT, SALT
```

这个函数以`crypt(3)`的方式计算一个字符串的单向散列。这对于检查口令文件来查看是否有糟糕的口令<sup>注4</sup>有些帮助，不过你真正希望的是人们一开始就不要增加不好的口令。

`crypt`是一个单向函数，这有点像打碎鸡蛋来做一个蛋饼。除了穷尽式强力猜测，没有（已知的）方法可以对一个加密的口令解密。

验证一个已有的加密字符串时，应当使用这个加密文本作为`SALT`（类似于`crypt($plain, $crypteq) eq $crypteq`）。这就允许你的代码使用标准`crypt`（以及更奇特的实现）。

选择一个新的`SALT`时，至少需要创建一个随机的两字符串，其字符来自于父集合`[./0-9A-Za-z]`（如`join "", (".", "/", 0..9, "A".."Z", "a".."z")[rand 64, rand 64]`）。`crypt`较早的实现只需要`SALT`的前两个字符，不过现在认为只提供前两个字符的代码不可移植。参见你的本地`crypt(3)`手册页来得到更多详细信息。

下面是一个例子，确保运行这个程序的人知道他自己的口令：

```

$pwd = (getpwuid ($<))[1]; # 假设在Unix上

system "stty -echo"; #或利用CPAN上的Term::ReadKey
print "Password: ";
chomp($word = <STDIN>);
print "\n";
system "stty echo";

if (crypt($word, $pwd) ne $pwd) {
    die "Sorry...\n";
} else {
    say "ok";
}

```

当然，如果有人问你口令，你就输入你的口令，这样做是不明智的。

`shadow`口令文件比传统的口令文件稍微安全一些，而且你必须是超级用户才能访问这些

注4： 只允许那些没有不良企图的人这样做。

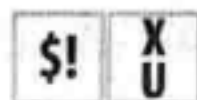


shadow口令文件。因为很少有程序需要在这么强的权限下运行，可以让程序维护自己独立的验证系统，把crypt字符串存储在非/etc/passwd或/etc/shadow的另外一个文件中。

crypt函数不适合加密大量数据，尤其是你无法得到返回的信息。可以查看你喜欢的CPAN镜像上的Crypt::\*、Digest::\*和PGP::\*目录，了解一些可能有用的模块。

如果对一个Unicode字符串使用crypt，这个字符串中可能包含一些码点超过255的字符，Perl会尝试将这个字符串复制到一个8位字节字符串，然后再对这个副本调用crypt。如果可以这样做，那很好。如果不行，crypt会产生一个异常。

## dbmclose

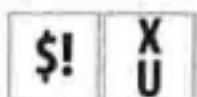


Dbmclose *HASH*

这个函数断开一个DBM（数据库管理）文件和一个散列之间的绑定。

dbmclose实际上就是一个提供了适当参数的untie调用，这个函数是为了支持与Perl较早版本的向后兼容。

## dbmopen



dbmopen *HASH, DBNAME, MODE*

这会把一个DBM文件绑定到一个散列（也就是一个关联数组）。DBM由一组C库例程组成，这些例程允许通过一个散列算法随机访问记录。*HASH*是散列的名字（包括%）。*DBNAME*是数据库的名字（没有.dir或.pag扩展名）。如果数据库不存在，而且指定了一个合法的*MODE*，会以*MODE*（经umask修改）指定的保护模式创建这个数据库。如果数据库不存在，要避免创建数据库，可以指定*MODE*为undef，如果函数无法找到一个现有的数据库，就会返回false。在dbmopen之前赋给散列的值不可访问。

dbmopen函数实际上就是一个提供了适当参数的tie调用，这个函数是为了支持与Perl较早版本的向后兼容。Dbmopen的返回值与你自己调用tie得到的返回值是一样的：成功时会返回绑定的对象，失败时返回false。通过直接使用tie接口，或者在调用dbmopen之前加载适当的模块，可以控制要使用哪个DBM库。下面这个例子可以在一些系统上运行，这些系统上DB\_File的版本类似于Netscape浏览器中的版本：

```
use DB_File;
dbmopen(%NS_Hist, "$ENV{HOME}/.netscape/history.dat", undef)
    || die "Can't open netscape history file: $!";

while (($url, $when) = each %NS_Hist) {
    next unless defined($when);
    chop ($url, $when); # 删除末尾的null字节
    printf "Visited %s at %s.\n", $url,
```

```
        scalar(localtime(unpack("V", $when)));
    }
}
```

如果你没有DBM文件的写访问权限，那么只能读散列变量而不能设置。如果你想测试你是否能写这些文件，可以使用一个文件测试操作符如`-w $file`，或者可以尝试在`eval {}`中设置一个虚拟散列记录，它会捕获异常。

对于很大的DBM文件，`keys`和`values`等函数会返回超大的列表值。你可能更愿意使用`each`函数来迭代处理大的DBM文件，这样就不用一次将整个文件都加载到内存中。

绑定到DBM文件的散列与你使用的DBM包类型有同样的限制，包括在一个桶中可以放入多少数据等等。如果坚持使用短的键和值，这一般没有问题。参见`DB_File`。

另外要记住的一点是，很多现有的DBM数据库包含以`null`结尾的键和值，因为它们是按C程序的考虑建立的。Netscape历史文件和老的`sendmail`别名文件就是这样的例子。要取出一个值并从这个值删除`null`，只需要使用`"$key\0"`。

```
$alias = $aliases{"postmaster\0"};
$alias =~ s/\0\z//; # 删除null
```

从v5.8.4开始，标准`DBM_Filter`模块可以自动为你处理以`null`结尾的字符串的问题。

```
use DB_File;
$db = dbmopen(%aliases, "/etc/mail/aliases", undef)
    || die "can't dbmopen /etc/mail/aliases: $!";
$db->Filter_Push("null");
$alias = $aliases{"postmaster"};
print "postmaster is aliased to $alias\n";
```

这个策略对于在句柄上增加一个`utf8`过滤器也很有用。参见第6章中的一个例子，它展示了如何使用Unicode作为DBM文件的键和值。

目前没有内置的方法来锁定一个通用DBM文件。有些人可能认为这是一个bug。`GDBM_File`模块在努力提供以整个文件为粒度的锁定机制。如果有疑问，最好使用一个单独的锁文件。

## defined

\$\_

```
defined EXPR
defined
```

这个函数返回一个布尔值，指示`EXPR`是否是一个已定义的值。我们处理的大多数数据都是已定义的，不过如果一个标量未包含合法的字符串、数值或引用值，则认为包含了未定义值，或简写为`undef`。将一个标量变量初始化为某个特定的值，实际上就是定义这个标量变量，它会一直保持为已定义，直到你为它赋一个未定义值，或者对这个变量显式地调用`undef`函数。

很多操作在特殊条件下会返回`undef`，如到达文件末尾，使用一个未初始化的变量的值，或者出现一个操作系统错误等等。由于`undef`就是一种`false`值，简单的布尔测试无法区分`undef`、数值0、`null`串和单字符串“0”，所有这些都等于`false`。如果使用的操作符可能返回一个真正的`null`字符串，可以利用`defined`函数来区分未定义的`null`字符串和已定义的`null`字符串。下面是一个代码片段，可以用来测试散列中的一个标量值：

```
print if defined $switch{D}; .
```

在类似这样的散列元素上使用时，`defined`会指出这个值是否已定义，而不论这个键在散列中是否有相应的记录。可能有一个键而相应的值未定义。不过，键本身是存在的。可以用`exists`确定散列键是否存在。

下一个例子中，我们利用了这样一个约定：即一些操作在处理完数据时会返回未定义的值（这里假设不存在包含`undef`的元素）：

```
print "$val\n" while defined($val = pop(@ary));
```

在这个例子中，我们对`getpwent`函数做了同样的测试，这个函数用于获取有关系统用户的信息。

```
setpwent();
while (defined($name = getpwent())) {
    say "<<$name>>";
}
endpwent();
```

从系统调用返回的错误也可以做这个测试，这些错误可能会合法地返回一个`false`值：

```
die "Can't readlink $sym: $!"
unless defined($value = readlink $sym);
```

你还可以使用`defined`来查看一个子例程是否已定义。这样可以避免当子例程不存在时出问题（或者子例程已经声明但没有给出定义）：

```
indir("funcname", @arglist);
sub indir {
    my $subname = shift;
    no strict "refs"; # 所以我们可以间接地使用subname
    if (defined &$subname) {
        &$subname(@_); #或$subname->(@_);
    }
    else {
        warn "Ignoring call to invalid function $subname";
    }
}
```

不过，即使是未定义的子例程，仍可以调用，因为它的包可能有一个`AUTOLOAD`函数，可以处理该包中未定义函数的调用。



在聚合类型（散列和数组）上使用defined的做法已经废弃。原先可以用它来报告这个聚合类型是否已经分配了内容。实际上，完全可以使用一个简单的布尔测试查看数组或散列是否包含元素：

```
if (@an_array) { print "has array elements\n" }
if (%a_hash) { print "has hash members\n" }
```

在一个散列元素上使用时，会告诉你这个值是否已定义，而不论这个键在散列中是否存在。可以用exists检查键是否存在。

参见undef和exists。

## delete

delete *EXPR*

这个函数从指定的散列或数组中删除一个元素（或者一个元素片段）。如果你想删除一个文件，参见unlink。删除的元素通常按指定的顺序返回，不过对于绑定变量（如DBM文件），则不能保证这个行为。删除操作之后，exists函数对于已经删除的键或索引会返回false（相比之下，调用undef函数之后，exists函数仍返回true，因为undef函数只是取消了元素值的定义，并没有删除这个元素本身）。

从%ENV散列删除元素会改变环境。如果一个散列绑定到一个（可写）DBM文件，从这个散列删除元素也会从该DBM文件删除相应的记录。

从数组删除元素会导致指定位置的元素回归到一种完全未初始化的状态，但是它不会补全这个空隙，否则会改变后面所有记录的位置。如果要补上这个空隙，需要使用splice。不过，如果删除数组中的最后一个元素，数组大小会减1或者更多，这取决于倒数第二个已有元素的位置（如果存在这个元素）。

在数组值上调用delete的做法已经废弃，在Perl的将来某个版本中可能会完全删除。

如果最后的操作是一个散列或数组查找，*EXPR*可能会非常复杂：

```
# 建立散列数组的数组
$dungeon[$x][$y] = \%properties;

# 从散列删除一个属性
delete $dungeon[$x][$y>{"OCCUPIED"};

# 从散列一次删除3个属性
delete @{$dungeon[$x][$y] }{ "OCCUPIED", "DAMP", "LIGHTED" };

# 从数组删除%properties的引用
delete $dungeon[$x][$y];
```

下面这个简单的例子从%hash删除所有值，但效率很低：

```
for my $key (keys %hash) {
    delete $hash{$key};
}
```

下面的效果是一样的：

```
delete @hash{keys %hash};
```

以上这两种做法的速度都比不上赋空列表或取消定义：

```
%hash = ();          # 完全清空%hash
undef %hash;          # 忘记%hash曾经存在过
```

对于数组也是类似的：

```
for my $index (0 .. $#array) {
    delete $array[$index];
}
```

以及：

```
delete @array[0 .. $#array];
```

都比下面的做法效率低：

```
@array = ();          # 完全清空@array
undef @array;          # 忘记@array曾经存在过
```

`delete local EXPR`构造可以将数组或散列元素的删除局部化，仅限于当前块。在这个块退出之前，局部删除的元素会临时不再存在。

## die

\$@

```
die LIST
die
```

在`eval`之外，这个函数会把`LIST`的联接值打印到`STDERR`并退出，退出时返回`$!`（C库的`errno`变量）的当前值。如果`$!`为0，则带`($? >> 8)`的值退出，这是从`system`、`wait`、管道`close`或者``command``最后俘获的子进程的状态。如果`($? >> 8)`为0，退出时则返回255。

在`eval`中，函数将`$@`变量设置为将要生成的错误消息，然后中止`eval`，这会返回`undef`。因此`die`函数可以用于生成命名异常，由程序的更高层捕获。参见本章后面的`eval`。

如果`LIST`是一个单对象引用，就认为这个对象是一个异常对象，将不加修改地返回，作为`$@`中的异常（见下面的介绍）。

如果`LIST`为空而且`$@`已经包含一个字符串值（通常来自之前的一个`eval`），会为此值追加“`\t...propagated`”之后再重用。这对于传播（再抛出）异常很有用：

```
eval { ... };
die unless $@ =~ /Expected exception/;
```

如果`LIST`为空而且`$@`已经包含一个异常对象，会调用这个对象的`$@->PROPAGATE`方法，并另外提供文件和行号参数来确定是否应当传播这个异常，用其返回值替换`$@`中的值。也就是说，就像调用了`$@ = eval { $@->PROPAGATE(__FILE__, __LINE__) }`一样。

如果`LIST`为空而且`$@`为空，则使用字符串“Died”。如果一个未捕获的异常导致解释器退出，则用以下伪代码由`!`和`?`的值确定退出码：

```
exit $!if $!;           # 错误号
exit $? >> 8 if $? >> 8; # 子进程退出状态
exit 255;               # 最后一招
```

其目的是将与原因有关的尽可能多的信息压缩到系统退出码的有限空间中。不过，由于任何系统调用都可以设置`!`，`die`使用的退出码的值可能是不可预测的，所以除非它不为0，否则不要依赖于这个值。

如果`LIST`的最终值不是以一个换行符结尾（而且没有传入一个异常对象），会把当前脚本文件名、行号和输入行数（如果有）追加到这个消息后面，另外再追加一个换行符。提示：如果追加类似“at scriptname line 123”的字符串，可以再为消息另外追加“，stopped”，这会让它更有意义。假设你要运行脚本`canasta`，考虑以下两种不同的退出方法的区别：

```
die "/usr/games is no good";
die "/usr/games is no good, stopped";
```

它们会分别生成以下结果：

```
/usr/games is no good at canasta line 123.
/usr/games is no good, stopped at canasta line 123.
```

如果你希望错误消息报告文件名和行号，可以使用`__FILE__`和`__LINE__`特殊token（不会在字符串中内插）：

```
die sprintf qq("%s" line "%s", phooey on you!\n),
__FILE__, __LINE__;
```

这会生成以下输出：

```
"canasta", line 38, phooey on you!
```

还有一个风格问题，考虑下面两个等价的例子：

```
die "Can't cd to spool: $!" unless chdir "/usr/spool/news";

chdir("/usr/spool/news") || die "Can't cd to spool: $!"
```

由于重要的部分是`chdir`，所以第二种形式更可取。



调用`die`时还可以提供一个引用参数，如果在`eval`中捕获了这个异常，`$@`就会包含这个引用。这允许更复杂的异常处理，可以使用对象维护有关异常的任意状态。与用正则表达式匹配`$@`的特定字符串值相比，这种机制有时更可取。因为`$@`是一个全局变量，`eval`可能在对象实现中使用，要当心分析错误对象时不要替换全局变量中的引用。最简单的做法是在所有处理之前先建立引用的一个局部副本。下面给出一个例子：

```
use Scalar::Util "blessed";

eval { WHATEVER; die Some::Module::Exception->new( FOO => "bar" ) };
if (my $eval_err = $@) {
    if (blessed($eval_err) && $eval_err->isa("Some::Module::Exception")) {
        # 处理Some::Module::Exception
    }
    else {
        # 处理所有其他异常
    }
}
```

对于未捕获的异常消息，在显示这些消息之前Perl会先完成字符串化，因此你可能要重载异常对象的字符串化操作。有关的详细内容参见第13章。

通过将`$SIG{__DIE__}`设置为一个要运行的函数，可以安排在`die`之前运行这个函数。这会基于错误文本调用关联的处理器，通过再次调用`die`改变错误消息（如果希望这样）。只有最成熟的向导工具才会尝试这种魔法，而且很少能存活下来。

参见`eval`、`exit`、`warn`、`%SIG`、`warnings pragma`和`Carp`模块。

## do (block)

`do BLOCK`

`do BLOCK`形式执行`BLOCK`中的语句序列，并返回块中最后计算的表达式的值。由一个`while`或`until`语句修饰符修饰时，Perl会在测试循环条件之前执行一次`BLOCK`（对于其他语句，循环修饰符会先测试条件）。`do BLOCK`本身不计为一个循环，所以不能用循环控制语句`next`、`last`或`redo`离开或重启这个块。解决办法参见第4章的“裸块作为循环”一节。

## do (file)

`do FILE`

`do FILE`形式使用`FILE`的值作为一个文件名，并把这个文件的内容作为Perl脚本执行。其主要用法是（或者原来的用法是）包含Perl子例程库中的子例程，所以：

```
do "stat.pl";
```



就类似于：

```
scalar eval `cat stat.pl`; # 在Windows上为`type stat.pl`
```

只不过`do`更高效、更简洁，可以跟踪当前文件名来得到错误消息，搜索`@INC`数组中所列的目录，如果找到文件还可以更新`%INC`（参见第25章）。还有一点区别，用`do FILE`执行的代码不能看到外围作用域中的词法作用域变量，而`eval FILE`中的代码可以。不过，有一点是相同的，每次调用时都会重新解析文件，所以你可能不想在循环中这么做，除非文件名本身在每次循环迭代时会改变。

如果`do`不能读文件，它会返回`undef`，并把`$!`设置为错误。如果`do`可以读文件但是无法编译，则返回`undef`，并在`$@`中设置一个错误消息。如果文件成功编译，`do`返回最后计算的表达式的值。

要包含库模块（必须有一个`.pm`后缀），最好使用`use`和`require`操作符来完成，这会完成错误检查，而且如果有问题还会生成一个异常。它们还能提供其他好处：避免重复加载，有助于面向对象编程，另外可以为编译器提供有关函数原型的提示。

不过，`do FILE`对于读取程序配置文件等工作还是有用的。可以如下手动地检查错误：

```
# 读入配置文件：先是系统配置，然后是用户配置
for $file ("/usr/share/proggie/defaults.rc",
          "$ENV{HOME}/.someprogrc")
{
    unless ($return = do $file) {
        warn "couldn't parse $file: $@" if $@;
        warn "couldn't do $file: $!"    unless defined $return;
        warn "couldn't run $file"       unless $return;
    }
}
```

长时间运行的守护进程可能会定期检查其配置文件的时间戳，如果文件在它最后一次读入之后有改变，守护进程可以使用`do`重新加载这个文件。这个工作用`do`完成比用`require`或`use`完成更简洁。

## do (subroutine)

`$@`

```
do SUBROUTINE(LIST)
```

`do SUBROUTINE(LIST)`是一种已经废弃的子例程调用。如果`SUBROUTINE`未定义会产生一个异常。参见第7章。

## dump

```
dump LABEL
dump
```

这个函数会导致立即内核转存。它的主要作用是，程序开始初始化所有变量之后，可以使用undump程序（未提供）将内核转储转换为一个可执行的二进制文件。执行这个新的二进制文件时，首先会执行一个goto LABEL（这会有goto的所有限制）。可以认为它是一个可以实现中间内核转储和程序重生的goto。如果省略LABEL，程序从头重启。警告：转存时打开的任何文件在程序重生时都不会打开，这可能会让Perl有些混乱。参见第17章中的-u命令行选项。

这个函数现在已经基本过时，部分原因在于，一般情况下很难把一个内核文件转换为一个可执行文件，另外还有一个原因：有很多可以生成可移植字节码和可编译C代码的不同的编译器后端，它们已经超越了这个函数。不过，管理Perl编译器项目（这是指在CPAN托管的perlcc及相关程序）的人报告称dump和undump支持可能很快会恢复。

如果想使用dump加快程序的执行速度，可以查看第21章关于效率问题的讨论，另外可以参见第16章对Perl原生代码生成器的讨论。还可以考虑自动加载和自加载，至少这会让程序看起来运行得更快。

## each



```
each Hash  
each ARRAY  
each EXPR
```

这个函数会单步处理一个散列，一次处理一个键/值对。在列表上下文中时，each返回一个包含两个元素的列表，它由散列中下一个元素的键和值组成，所以可以利用each函数迭代处理散列。在标量上下文中调用时，each只返回散列中下一个元素的键。如果已经完全读入散列，会返回空列表，赋值为这个空列表时，在标量上下文中（如循环测试）会生成一个false值。在此之后的下一个each调用会重新开始迭代。典型用法如下，这里使用预定义的%ENV散列：

```
while (($key,$value) = each %ENV) {  
    say "$key=$value";  
}
```

在内部，散列采用一种看上去随机的顺序维护它自己的记录。each函数迭代处理这个序列，因为每个散列会记住最后返回哪个记录。将来的Perl版本中这个序列的具体顺序可能会改变，不过肯定与keys（或values）函数处理相同散列（未修改）时生成的顺序是一样的。出于安全性考虑，同一个程序不同次运行时这个顺序会有所不同。

Perl为各个散列维护了一个迭代器，程序中的所有each、keys和values函数调用都共享这个迭代器，可以读取散列中的所有元素来重置迭代器，或者通过计算keys %hash或values %hash来重置。如果迭代处理散列时增加或删除了散列元素，其结果没有明确的定义，可能会跳过某些记录，或者重复某些记录。



从v5.12开始，还可以使用数组作为`each`的参数。数组的键就是它的索引。不同于散列，会按键（数组索引）的升序顺序返回键值对。

从v5.14开始，`each`可以取一个未祝福的散列或数组的引用作为参数，这会自动解引用。`each`的这个方面被认为是试验性的。具体的行为在Perl将来的版本中有可能改变。

```
while (($key,$value) = each %hashref) { ... }
```

参见`keys`、`values`和`sort`。

## eof



```
eof FILEHANDLE  
eof()  
eof
```

如果对`FILEHANDLE`的下一个读操作返回文件末尾（end-of-file），或者如果`FILEHANDLE`未打开，这个函数将返回`true`。`FILEHANDLE`可以是一个表达式，其值可以真正的文件句柄，或者是某种文件句柄对象的一个引用。如果没有参数，`eof`会返回最后一个文件读操作的文件末尾状态。带空括号()`eof()`要检查`ARGV`文件句柄（最常见的是`<>`中的空文件句柄）。因此，在一个`while (<>)`循环中，有小括号的`eof()`只会检测一组文件中最后一个文件的文件末尾状态。可以使用`eof`（不带小括号）在`while (<>)`循环中测试每一个文件。例如，下面的代码在最后一个文件的最后一行前面插入短横线：

```
while (<>) {  
    if (eof()) {  
        say "-" x 30;  
    }  
    print;  
}
```

下面这个脚本对每个输入文件重置行数：

```
# 对每个输入文件重置行数  
while (<>) {  
    next if /^\\s*#/;          # 跳过注释  
    print "$_\\t$ ";  
} continue {  
    close ARGV if eof;        # Not eof()!  
}
```

与`sed`程序中的“\$”类似，`eof`会显示行数范围。下面的脚本可以打印从`/pattern/`到各个输入文件末尾之间的行：

```
while (<>) {  
    print if /pattern/ .. eof;  
}
```

在这里，触发器操作符（`..`）为每一行进行模式匹配。模式匹配前，操作符返回`false`。最终匹配时，操作符开始返回`true`，从而打印这些行。`eof`操作符最终返回`true`时（在所检查的文件的最后），触发器操作符被重置，并对`@ARGV`中的下一个文件再次开始返回`false`。

警告：`eof`函数读入一个字节，然后用`ungetc(3)`把它压回到输入流中，所以在交互式上下文中没有什么用处。有经验的Perl程序员很少使用`eof`，因为各种输入操作符在`while`-循环条件中已经表现得很好。参见第4章`foreach`描述中的例子。

## eval



```
eval BLOCK
eval EXPR
eval
```

`eval`关键字在Perl中有两个不同用途，但彼此相关。这两种用途分别用两种不同的语法形式表示，`eval BLOCK`和`eval EXPR`。第一种形式捕获运行时异常（错误），如果不捕获这些异常将是致命的，类似于C++或Java中的“try块”构造。第二种形式在运行时动态编译并执行小段代码，也会像第一种形式一样（很方便地）捕获异常。不过第二种形式比第一种形式运行慢得多，因为它每次都必须解析字符串。另一方面，这种形式更通用。不论使用哪一种形式，`eval`都是Perl中完成异常处理的首选方法。

这两种形式的`eval`都返回最后计算的表达式的值，与子例程一样。类似的，可以使用`return`操作符从`eval`的中间返回一个值。提供返回值的表达式可以在`void`上下文、标量上下文或列表上下文中计算，这取决于`eval`本身的上下文。关于如何确定计算上下文，更多内容参见`wantarray`。

如果有一个可捕获的错误（包括`die`操作符生成的任何错误），`eval`会返回`undef`，并把错误消息（或对象）放在`$@`中。如果没有错误，`$@`会设置为`null`串，所以以后完全可以测试`$@`来确定是否有错误。一个简单的布尔测试就足够了：

```
eval { ... };      # 捕获运行时错误
if ($@) { ... }    # 处理错误
```

`eval BLOCK`形式会在编译时检查语法并编译，所以它在运行时与任何其他块一样高效（熟悉慢速的`eval EXPR`形式的人有时会被这个问题搞糊涂）。由于`BLOCK`与周围代码一同编译，这种形式的`eval`不能捕获语法错误。

`eval EXPR`形式之所以可以捕获语法错误，原因在于它在运行时解析代码（如果解析不成功，它会像平常一样把解析错误放在`$@`中）。如果解析成功，会执行`EXPR`的值，就好像它是一个小Perl程序。代码在当前Perl程序的上下文中执行，这说明它可以从外围作用域看到所有外围词法作用域变量，而且在`eval`完成之后，所有非局部变量设置仍有效，所有子例程或格式定义也仍然有效。`Eval`的代码会处理为一个块，所以在`eval`中声明的所有局部

作用域变量只能持续到eval结束（参见my和local）。与块中的任何代码一样，最后一个分号不是必需的。

下面是一个简单的Perl shell脚本。它提示用户输入一个包含任意Perl代码的字符串，编译并执行这个字符串，然后打印可能生成的错误：

```
print "\nEnter some Perlcode: ";

while (<STDIN>) {
    eval;
    print $@;
    print "\nEnter some more Perlcode: ";
}
```

下面给出一个重命名（rename）程序，这里使用一个Perl表达式对文件成批重命名：

```
#!/usr/bin/perl
# 重命名，改变文件名
$op = shift;
for (@ARGV) {
    $was = $_;
    eval $op;
    die if $@;
    # 下一行调用内置函数
    # 而不是同名的脚本
    rename($was,$_ ) unless $was eq $_;
}
```

可以如下使用这个程序：

```
% rename 's/\.orig$//'          *.orig
% rename 'y/A-Z/a-z/ unless /^Make/' *
% rename '$_ .= ".bad"'          *.f
```

由于eval会捕获错误（否则会是致命错误），它对于判断是否实现了某些特定的特性（如fork或symlink）很有用。

因为eval BLOCK在编译时检查语法，所以语法错误会较早报告。因此，如果你的代码是不变的，而且eval EXPR和eval BLOCK都能满足你的要求，在这种情况下更倾向于BLOCK形式。例如：

```
# 使除0错误为非致命错误
eval { $answer = $a / $b };    warn $@ if $@;

#作用相同，但是如果运行多次，效率相对较低
eval '$answer = $a / $b';    warn $@ if $@;

# 一个编译时语法错误（未捕获）
eval { $answer = };          #不正确

# 一个运行时语法错误
eval '$answer =';            # sets $@
```

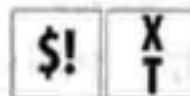


在这里，只有当BLOCK中的代码是合法Perl代码时才能通过编译阶段。EXPR中的代码直到运行时才会检查，所以它在运行时之前不会导致错误。

eval BLOCK的块不算作一个循环，所以不能用循环控制语句next、last或redo离开或重启这个块。

DB包中执行的eval字符串不会看到外围词法作用域，而会看到调用它的第一个非DB代码的作用域。正常情况下不需要担心这一点，除非你要写一个Perl调试器。

## exec



```
exec PATHNAME LIST
exec LIST
```

exec函数会终止当前程序，并执行一个外部命令，而且不会返回！要想在命令完成后返回程序，应当使用system而不是exec。只有当命令不存在，而且是直接执行命令而不是通过系统的命令shell（见下面的讨论）来执行，exec函数才会失败并返回false。

如果只有一个标量参数，会检查这个参数是否有shell元字符。如果找到元字符，整个参数会传递到系统的标准命令解释器（在Unix下是/bin/sh）。如果没有元字符，参数则分解为单词，并直接执行，因为从效率的角度讲，这样可以避开shell处理的开销。如果程序不存在，它还允许你对错误恢复有更多控制。

如果在LIST中有多个参数，或者如果LIST是一个包含多个值的数组，就不会使用系统shell。这也会避开命令的shell处理。参数中是否存在元字符并不影响这个列表触发的行为，这使它成为需要考虑安全性的程序的首选形式，这些程序不希望暴露在shell转义可能带来的注入攻击之下。

这个例子会使当前运行的Perl程序把自己替换为echo程序，然后打印当前的参数列表：

```
exec "echo", "Your arguments are: ", @ARGV;
```

由这个例子可以看到，可以使用exec执行一个管线而不只是单个程序。

```
exec "sort $outfile | uniq"
|| die "Can't do sort/uniq: $!";
```

正常情况下，exec不会返回。如果它确实返回了，也总是返回false，应当检查\$!来查看哪里出了问题。在很老版本的Perl中（v5.6之前），exec（和system）不会刷新输出你的输出缓冲区，所以需要在的一个或多个文件句柄上设置\$|来启用命令缓冲，以避免由于exec而丢失输出，或者由于system得到顺序有误的输出。

要求操作系统在现有的一个进程中执行一个新程序时（如Perl的exec函数所做的），需要告诉系统要执行的程序的位置，还要告诉这个新程序（通过其第一个参数）将以什么名字

来调用它。习惯上，你给出的名字只是程序位置的一个副本，不过不一定非得如此，因为C语言级别上有两个单独的参数。如果不是一个副本，就会得到一个奇怪的结果，新程序认为它以某个名字运行，而这个名字与程序所在的具体路径名完全不同。通常这对于当前的程序并不重要，但是有些程序确实关心这一点，而且会根据它们所认为的名字表现出不同的“个性”。例如，*vi*编辑器会查看它是作为“*vi*”还是作为“*view*”调用。如果作为“*view*”调用，会自动启用只读模式，就好像调用时有`-R`命令行选项一样。

这里就要用到`exec`的可选`PATHNAME`参数。从语法上讲，它就像`print`或`printf`的文件句柄一样进入间接对象槽。因此后面没有逗号，因为这不是参数表的一部分（从某种意义上讲，Perl采用了与操作系统相反的方法，它假设第一个参数是重要的参数，如果有不同，则允许你修改路径名）。例如：

```
$editor = "/usr/bin/vi";
exec $editor "view", @files # 触发只读模式
|| die "Couldn't execute $editor: $!";
```

与所有其他间接对象一样，还可以把包含程序名的简单标量替换为一个包含任意代码的代码块，这样可以把前面的例子简化为：

```
exec { "/usr/bin/vi" } "view", @files # 触发只读模式
|| die "Couldn't execute /usr/bin/vi: $!";
```

前面我们已经提到，`exec`会把离散的参数表作为一个指令来避开shell处理。不过，有一点可能还会让人有些困惑。`exec`（以及`system`）调用无法区分单个标量参数和只包含一个元素的数组。

```
@args = ("echo surprise"); # 列表中只有一个元素
exec @args                 # 仍有shell转义
|| die "exec: $!";         # 因为@args == 1
```

为了避免这种情况，可以使用`PATHNAME`语法，显式地重复第一个参数作为路径名，这会强制将其余的参数解释为一个列表，尽管其中只有一个元素：

```
exec { $args[0] } @args # 即使是只有一个参数的列表也是安全的
|| die "can't exec @args: $!";
```

第一个版本（即没有大括号的版本）会运行`echo`程序，传入“`surprise`”作为参数。第二个版本则不会，它会按字面运行一个名为`echo surprise`的程序，但找不到这个程序（希望如此），因而将`$!`设置为一个非0值来指示失败。

由于`exec`函数通常都紧跟在`fork`后使用，它假设应当跳过Perl进程终止时原本要发生的所有事情。对于`exec`，Perl不会调用`END`块，也不会调用与任何对象关联的任何`DESTROY`方法。否则，子进程将完成你本来希望父进程完成的清理工作（我们希望现实生活中能够这样）。



由于把`exec`当作`system`使用是一种很常见的错误，如果后面的语句不是`die`、`warn`或`exit`，而且假设你已经启用了警告，Perl就会向你发出警告（你确实已经启用了警告，是吧）。如果你确实希望`exec`后面是另外某个语句，可以使用以下任何一种风格来避免警告：

```
exec ("foo") || print STDERR "couldn't exec foo: $!";
{ exec ("foo") }; print STDERR "couldn't exec foo: $!";
```

如以上第二行所示，作为块中最后一个语句的`exec`调用可以免于警告。

Perl在执行`exec`之前，会尝试刷新输出所有为输出所打开的文件，不过这在某些平台上可能并不支持。为了能安全地这样做，可能需要设置`$|`（使用`English`模块则是`$AUTOFLUSH`），或者对所有打开的句柄调用`IO::Handle`的`autoflush`方法，来避免丢失输出。

需要说明，对于`exec`，会调用`END`块，也不会调用对象上的`DESTROY`方法。

参见`system`。

## exists

`exists EXPR`

给定一个指定某个散列元素的表达式，如果指定的散列元素已经初始化，这个函数返回`true`，即使这个元素的值未定义也没有影响。

```
print "True\n"    if $hash{$key};
print "Exists\n"  if exists $hash{$key};
print "Defined\n" if defined $hash{$key};
```

以前，`exists`还可以在数组元素上调用，但其本身的行为不太明显，往往与数组的`delete`用法紧密绑定。不过，对数组值调用`exists`的做法已经过时，很可能在Perl将来的版本中完全删除。

```
print "True\n"    if $array[$index];
print "Exists\n"  if exists $array[$index];
print "Defined\n" if defined $array[$index];
```

只有当元素已定义时才`exists`可能为`true`，而且只有元素存在时才可能已定义，不过反过来不一定成立。

如果最后一个操作是一个散列键或数组索引查找，`EXPR`可能非常复杂：

```
if (exists $hash{A}{B}{$key}) { ... }
```

尽管最后一个元素不会因为检查其存在性而突然出现，但中间的元素是有这个可能的。因



此，`$$hash{"A"}`和`$hash{"A"}->{"B"}`会突然存在。本质上讲，这不是`exists`本身的功能；任何使用了箭头的地方都可能发生（显式或隐式）这种情况：

```
undef $ref;
if (exists $ref->{"Some key"}) { }
print $ref; # 打印HASH(0x80d3d5c)
```

尽管“Some key”元素并没有出现，但之前原本未定义的`$ref`变量确实突然包含了一个匿名散列。这是一个让人惊讶的自动生成（autovivification）的例子，第一眼（甚至第二眼）看上去可能认为这不会在一个左值上下文中发生。这种行为很可能在将来的版本中得到修正。要绕开这个问题，可以嵌套调用：

```
if (
    $ref                &&
    exists $ref->[$x]     &&
    exists $ref->[$x][$y] &&
    exists $ref->[$x][$y]{$key} &&
    exists $ref->[$x][$y]{$key}[2] ) { ... }
```

如果`EXPR`是子例程名，而且这个子例程已经声明，即使该子例程尚未定义，`exists`函数也会返回true。下面只打印“Exists”：

```
sub flub;
print "Exists\n" if exists &flub;
print "Defined\n" if defined &flub;
```

对一个子例程名使用`exists`可能对于`AUTOLOAD`子例程很有用，它需要知道某个特定的包是否希望定义某个特定的子例程。这个包可以指示这一点，为此可以声明一个桩sub如`flub`，如上所示。

如果意外地使用一个子例程调用的返回值而不是子例程名作为`exists`的参数，这是一个错误。

```
exists &sub; # 正确
exists &sub(); # 错误：父进程会调用函数
```

## exit

```
exit EXPR
exit
```

这个函数将`EXPR`作为一个整数来计算，并以该值作为程序的最终错误状态立即退出。如果省略`EXPR`，函数会以0状态退出（表示“无错误”）。下面的代码片段允许用户键入x或X退出程序：

```
$ans = <STDIN>;
exit if $ans =~ /^[Xx]/;
```

如果有人想要捕获发生的错误，不要使用`exit`终止子例程。对此应当使用`die`，这可以用一个`eval`捕获。或者使用Carp模块的某个`die`的包装器，如`croak`或`confess`。

我们说`exit`函数会立即退出，不过这是一个赤裸裸的谎言。它会尽可能快地退出，但是首先它要调用所有已定义的END例程，完成退出时处理。这些例程不能中止退出，不过它们可以通过设置`$?`变量改变最终的退出值。类似地，如果一个类定义了一个`DESTROY`方法，在程序真正退出之前会代表这个类的对象调用这个方法。如果你确实需要绕过退出处理，可以调用POSIX模块的`_exit`函数来避免所有END和析构函数处理，如果POSIX不可用，可以使用`exec "/bin/false"`或类似的函数。

## exp

\$\_

```
exp EXPR
exp
```

这个函数返回 $e$ （自然对数的底数）的`EXPR`次幂。要得到 $e$ 的值，可以使用`exp(1)`。对于不同底数的一般指数运算，可以使用从FORTRAN“窃取”的`**`操作符：

```
use Math::Complex;
print -exp(1) ** (i * pi); # 打印1
```

## \_\_FILE\_\_

这是一个特殊token，返回出现这个token的文件的名字。参见第21章“用其他语言生成Perl”一节。

## fc

\$\_ T

```
fc EXPR
fc
```

这是v5.16新增的函数，这个函数返回`EXPR`的完全Unicode大小写转换（`casefold`）。这是实现大小写转换字符串中\F转义的内部函数。标题形式（`titlecase`）基于大写，但与大写形式有所不同，类似的，`foldcase`基于小写，但与小写形式也有所不同。在ASCII中，只有两个大小写形式（即大写和小写），而且它们之间存在一对一的映射，不过在Unicode中有3种形式，而且这3种形式之间存在一对多的映射。由于有太多的组合，无法每次都手动地检查，所以发明了第4种大小写形式，称为`foldcase`，这作为另外3种形式的一个常用的中间形式。它本身不是一种大小写，而只是一种大小写映射。

如果不考虑大小写，要比较两个字符串是否相等，可以这样做：

```
fc($a) eq fc($b)
```

在v5.16之前，如果要不区分大小写比较字符串，唯一可靠的方法就是利用/i模式修饰符，因为Perl对于不区分大小写的模式匹配使用大小写转换语义。了解到这一点，可以模拟相等性比较，如下所示：

```
sub fc_eq($$) {  
    my($a, $b) = @_;  
    return $a =~ /\A\Q$b\E\z/i;  
}
```

对于v5.16之前的版本，可以在CPAN上的Unicode::CaseFold模块中找到fc函数。要完成不区分重音符和不区分大小写的比较，可以使用eq或cmp方法并结合一个Unicode::Collate排序器对象，这会在其构造函数中传入level=>1，或者利用一个为本地化环境相等比较和排序顺序以类似方式构建的Unicode::Collate::Locale对象。参见第6章中“张冠李戴”和“Unicode文本比较与排序”小节。

## fcntl



fcntl FILEHANDLE, FUNCTION, SCALAR

这个函数调用你的操作系统的文件控制函数，如fcntl(2)手册页所述。在调用fcntl之前，可能首先要声明：

```
use Fcntl;
```

来加载正确的常量定义。

将根据FUNCTION读或写（或同时读写）SCALAR。SCALAR字符串值的指针将作为实际fcntl调用的第3个参数传入（如果SCALAR没有字符串值，但有一个数字值，这个值会直接传入，而不是传入字符串值的一个指针）。对于FUNCTION可取的更多常用值，参见Fcntl模块的描述。

如果在一个没有实现fcntl(2)的系统上使用fcntl函数，会产生一个异常。在支持fcntl(2)的系统上，可以做很多工作，如修改“执行时关闭”（close-on-exec）标志（如果你不想使用\$^F（\$SYSTEM\_FD\_MAX）变量）、修改非阻塞I/O标志、模拟lockf(3)函数，以及I/O等待时安排接收SIGIO信号。

下面给出一个例子，在系统级将一个名为REMOTE的文件句柄设置为非阻塞。这样一来，从一个管道、套接字或串行线读取数据时，如果没有可用数据，输入操作会立即返回，否则会阻塞。它还会导致正常情况下阻塞的输出操作返回一个失败状态（在这些情况下，可能还必须调整\$|）。

```
use Fcntl qw(F_GETFL F_SETFL O_NONBLOCK);  
  
$flags = fcntl(REMOTE, F_GETFL, 0)
```



```

        || die "Can't get flags for socket: $!";

$flags = fcntl(REMOTE, F_SETFL, $flags | O_NONBLOCK)
        || die "Can't set flags for socket: $!";

```

fcntl（和ioctl）的返回值如表27-1所示。

表27-1：fcntl的返回值

系统调用返回值	Perl返回值
-1	undef
0	字符串“0 but true”
任何其他结果	该数字

因此，成功时Perl会返回true，失败时返回false，不过可以很容易地确定操作系统返回的具体值：

```

$retval = fcntl(...) || -1;
printf "fcntl actually returned %d\n", $retval;

```

在这里，甚至字符串“0 but true”也会打印为0，这是由于%d格式的作用。这个字符串在布尔上下文中是true，在数值上下文中则是0。这允许你对返回值使用一个简单的|| die测试，而不是“歪斜的”版本// die（幸运的是，这样也能避免正常情况下可能对不正确的数值转换作出的警告）。

## fileno



fileno FILEHANDLE

这个函数返回文件句柄底层的文件描述符。如果这个文件句柄未打开，fileno返回undef。如果OS层没有真正的文件描述符，通过open（以一个引用作为第三个参数）与内存对象连接的文件句柄就可能出现这种情况，此时会返回-1。

文件描述符是一个很小的非负整数，如0或1，这与类似STDIN和STDOUT的文件句柄不同，它们都是符号。遗憾的是，操作系统不认识这些酷酷的符号。它只考虑用这些小文件号打开文件，尽管Perl通常会为你自动完成转换，不过有时还是需要知道具体的文件描述符。

所以，举例来说，fileno函数对于为select构建位图很有用，如果实现了syscall(2)，这个函数对于传递某些晦涩的系统调用也很有用。它还可以用来反复检查open函数提供的是不是你想要的文件描述符，以及确定两个文件句柄是否使用相同的系统文件描述符。

```

if (fileno(THIS) == fileno(THAT)) {
    say "THIS and THAT are dups";
}

```

如果`FILEHANDLE`是一个表达式，这个值会作为一个间接文件句柄，通常是这个文件句柄名，或者是某个类文件句柄对象的引用。

在程序的整个生命期中都不要依赖于Perl文件句柄和数值文件描述符的关联。如果一个文件关闭后重新打开，文件描述符可能会改变。Perl会努力确保当某些文件描述符的`open`失败时不会丢失这些文件描述符，不过这只对未超过特殊`$^F` (`$SYSTEM_FD_MAX`)变量当前值（默认为2）的文件描述符有效。尽管文件句柄`STDIN`、`STDOUT`和`STDERR`最初的文件描述符分别为0、1和2（Unix标准约定），不过如果你随意地关闭和打开这些文件句柄，这就可能改变。只要关闭之后总是立即重新打开文件，就不会有麻烦，可以认为它们的文件描述符为0、1和2。在Unix系统上，基本原则是总是尽可能选择最小的描述符，也就是你刚关闭的那个描述符。

## flock



`flock FILEHANDLE, OPERATION`

`flock`函数是Perl的可移植的文件锁定接口。它只锁定整个文件，而不是单个记录。这个函数会管理与`FILEHANDLE`关联的文件上的锁，成功时返回`true`，否则返回`false`。为了避免有可能丢失数据，Perl在锁定或解除锁定之前会刷新输出`FILEHANDLE`。Perl可以使用`flock(2)`、`fcntl(2)`、`lockf(3)`或另外某种特定于平台的锁定机制来实现其`flock`。如果这些函数都不可用，调用`flock`会生成一个异常。参见第15章中“文件锁定”一节。

`OPERATION`可以是`LOCK_SH`、`LOCK_EX`或`LOCK_UN`，可能与`LOCK_NB`相或（OR）。这些常量的取值通常是1、2、8和4，不过如果从`Fcntl`模块导入（单独导入，或者使用标记：`flock`成组地导入），也可以使用符号名。

`LOCK_SH`请求一个共享锁，所以通常用于读。`LOCK_EX`请求一个排他锁，所以通常用于写。`LOCK_UN`释放先前请求的一个锁，关闭文件也会释放这个文件上的所有锁。如果`LOCK_NB`位与`LOCK_SH`或`LOCK_EX`结合使用，`flock`会立即返回而不会等待一个不可用的锁。可以检查返回状态，来看你是否得到了所请求的锁。如果不使用`LOCK_NB`，可能会无限地等待所请求的锁。

`flock`还有一个不太明显但很常见的方面：它的锁只是建议性的。任意锁更灵活，但是没有强制锁那么有保证。这说明用`flock`锁定的文件可以由不使用`flock`的程序修改。等待红灯的车彼此能和平共处，不过看到红灯也不停的车就会带来问题。请文明驾驶。

`flock`的某些实现不能通过网络加锁。尽管从理论上讲这一点可以使用更特定的`fcntl`来完成，不过评审团（本来在20多年前已经隐退）还在争论这么做是否可靠（或者甚至是否能这么做）。

下面是用于Unix系统的一个邮箱附件程序，这里使用`flock(2)`锁定邮箱：

```

use Fcntl qw/:flock/; # 导入LOCK_*常量
sub mylock {
    flock(MBOX, LOCK_EX)
    || die "can't lock mailbox: $!";
    # 万一有人在我们等待时追加
    # stdio缓冲区将不同步
    seek(MBOX, 0, 2)
    || die "can't seek to the end of mailbox: $!";
}

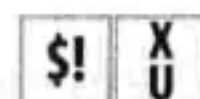
open(MBOX, ">> /usr/spool/mail/$ENV{USER}")
    || die "can't open mailbox: $!";

mylock();
say MBOX $msg, "\n";
close MBOX
    || die "can't close mailbox: $!";

```

在支持`flock(2)`系统调用的系统上，锁是通过`fork`调用继承的。其他实现没有这么幸运，有可能在`fork`之间丢失锁。参见第15章中“文件锁定”一节中的其他`flock`例子。

## fork



`fork`

这个函数通过调用`fork(2)`系统调用由一个进程创建两个进程。如果成功，这个函数将向父进程返回新的子进程的ID，并向子进程返回0。如果系统没有足够的资源来分配一个新进程，调用会失败，并返回`undef`。文件描述符（有时还有这些描述符上的锁）是共享的，所有其他方面会复制，或者至少看起来会复制。

在原来的版本中，未刷新输出的缓冲区在两个进程中都会保持这个状态（即不刷新输出），这说明你可能需要在程序较早前对一个或多个文件句柄设置`$|`，以避免重复输出。

要启动一个子进程，同时检查是否存在“不能派生子进程”错误，对此有一个近乎完美的方法：

```

use Errno qw(EAGAIN);
FORK: {
    if ($pid = fork) {
        # 这里是父进程
        # 子进程pid可以从$pid得到
    }
    elsif (defined $pid) { # 如果已定义，$pid为0
        # 这里是子进程
        # 父进程pid可以由getppid得到
    }
    elsif ($! == EAGAIN) {
        # EAGAIN是通常认为可以恢复的fork错误
        sleep 5;
    }
}

```



```

        redo FORK;
    }
    else {
        # 奇怪的fork错误
        die "Can't fork: $!";
    }
}

```

对于完成一个隐式`fork(2)`的操作，如`system`、反引号或者将一个进程作为文件句柄打开，这些措施是没有必要的，因为在为你完成`fork`时，如果出现暂时失败，Perl会自动重新尝试使用`fork`派生子进程。要注意，需要用一个`exit`；结束子进程代码，否则，子进程可能不小心退出代码块，而开始执行本来只应由父进程执行的代码。

如果只使用`fork`而不等待子进程，就会累积大量僵尸进程（这些进程已经退出，但它们的父进程没有等待这些子进程）。在一些系统上，可以通过将`$SIG{CHLD}`设置为“IGNORE”来避免这个问题。大多数情况下，必须用`wait`等待垂死的子进程。参见这一章中的“wait”来了解有关的例子，或者参见第15章中“信号”一节来了解SIGCHLD的更多信息。

如果派生的子进程继承了类似STDIN和STDOUT等系统文件描述符，这些子进程连接到一个远程管道或套接字，你可能必须在子进程中将它们重新打开到`/dev/null`。这是因为，即使父进程退出，但由于这些文件句柄副本的存在，子进程可能仍然存在。远程服务器（比如说一个CGI脚本，或者从一个远程shell启动的后台任务）看上去会挂起，因为它还在等待所有副本关闭。通过重新打开这些系统文件句柄指向其他设备，可以修正这个问题。

在大多数支持`fork(2)`的系统上，人们做了大量努力，使它极为高效（例如，对数据页使用写时复制技术），所以这也成为过去几十年实现多任务的主要模式。在与Unix不同的系统上，高效地实现`fork`函数不太可能（或者可能根本不能实现）。例如，尽管Perl在Microsoft系统上模拟了一个`fork`，但是对性能没有任何保证。幸运的话，可以使用`Win32::Process`模块。

Perl在派生子进程之前，会尝试刷新输出所有为输出打开的文件，不过在一些平台上并不支持这一点。为了保证安全，可能需要设置`$|`（使用English模块则是`$AUTOFLUSH`），或者对打开的句柄调用`IO::Handle`的`autoflush`方法，以避免重复输出。

如果只使用`fork`而不等待子进程，就会累积大量僵尸进程。在一些系统上，可以通过将`$SIG{CHLD}`设置为“IGNORE”来避免这个问题。另外参见第15章了解派生子进程和僵尸进程的更多信息。

需要说明，如果派生的子进程继承了类似STDIN和STDOUT等系统文件描述符，这些子进程连接到一个管道或套接字，远程服务器（比如说一个CGI脚本，或者从一个远程shell启动

的后台任务)并不认为你已经结束,即使已经退出了父进程。如果存在这个问题,需要重新打开这些系统文件句柄到/dev/null。

## format

```
format NAME =  
    picture line  
    value list  
    ...  
.
```

这个函数声明一个形象行的命名序列(及相关的值),供write函数使用。如果省略了NAME,名字则默认为STDOUT,这正好是STDOUT文件句柄的默认格式名。类似于sub声明,由于这是发生在编译时的一个包全局声明,要求值列表中使用的任何变量在格式声明时可见。也就是说,词法作用域变量必须已经在文件前面声明,而动态作用域变量只需要在调用write时设置。下面是一个例子(这里假设我们已经计算了\$cost和\$quantity):

```
my $str = "widget";                # 词法作用域变量  
  
format Nice_Output =  
Test: @<<<<<<< @|||| @>>>>>>  
      $str, $%, '$' . int($num)  
.  
  
local $~ = "Nice_Output";          # 选择我们的格式  
local $num = $cost * $quantity;     # 动态作用域变量  
  
write;
```

类似于文件句柄,格式名是存在于符号表(包)中的标识符,可以由包名完全限定。在一个符号表记录的类型团中,格式在其自己的命名空间中,与文件句柄、目录句柄、标量、数组、散列和子例程的命名空间不同。不过,与另外这6种类型类似,名为Whatever的格式也会受到\*Whatever类型团上的local声明的影响。换句话说,格式只是包含在类型团中的另外一个部件,独立于其他部件。

第26章“形象格式”一节包含了关于这些用法的很多细节和例子。第25章介绍了内部格式特定的变量,利用English和IO::Handle模块可以更容易地访问这些变量。

## formline

```
formline PICTURE, LIST
```

这是format使用的一个内部函数,不过你也可以自行调用这个函数。它总是返回true。这个函数会根据PICTURE的内容格式化一个值列表,将输出放入格式输出累加器\$^A(或者如果使用English模块则是\$ACCUMULATOR)。最后,完成一个write时,\$^A的内容写至某个文件句柄,不过你也可以自行读取\$^A,然后把\$^A设置回"。通常格式会对表格中的每

一行完成一个formline，不过formline函数本身并不关心PICTURE中嵌入多少个换行符。这说明~ 和~~ token会把整个PICTURE当作是一行。因此可能需要使用多个formline来实现一个记录格式，格式编译器在内部就是这样做的。

如果在形象周围加双引号，一定要当心，因为@字符可能要用来表示一个数组名的开头。参见第26章“形象格式”来了解它的用法。

## getc



```
getc FILEHANDLE  
getc
```

这个函数返回与FILEHANDLE关联的输入文件的下一个字符。如果到达文件末尾，或者如果遇到一个I/O错误，这个函数会返回undef。如果省略FILEHANDLE，函数将从STDIN读取。

这个函数速度有点慢，不过有时对于从键盘输入单字符会很有用（假设你能够做到不缓冲键盘输入）。这个函数从标准I/O库请求未缓冲的输入。遗憾的是，标准I/O库并不太标准，无法提供一种可移植的方法来告诉底层操作系统将未缓冲的键盘输入提供给标准I/O系统。为此，你必须更聪明一些，并采用一种依赖于操作系统的方式。在Unix下，可以写为：

```
if ($BSD_STYLE) {  
    system "stty cbreak </dev/tty >/dev/tty 2>&1";  
} else {  
    system "stty", "-icanon", "eol", "\001";  
}  
  
$key = getc;  
  
if ($BSD_STYLE) {  
    system "stty -cbreak </dev/tty >/dev/tty 2>&1";  
} else {  
    system "stty", "icanon", "eol", "^@"; # ASCII NUL  
}  
print "\n";
```

这个代码把终端上输入的下一个字符放在字符串\$key中。如果你的stty程序有类似cbreak的选项，那么只要\$BSD\_STYLE为true就需要使用以上代码。否则，如果stty程序没有cbreak选项，则要在\$BSD\_STYLE为false时使用这个代码。如何确定stty(1)的选项，这作为一个练习留给读者来完成。

POSIX模块在声称符合POSIX的系统上使用POSIX::getattr函数提供了一个更可移植的版本。另外参见离你最近的CPAN网站的Term::ReadKey模块，来获得一种更可移植、更灵活的方法。对于ungetc函数，可以使用IO::Handle类中的相应方法。



## getgrent



```
getgrent
setgrent
endgrent
```

这些子例程迭代处理你的`/etc/group`文件（或者其他人的`/etc/group`文件，如果它来自另外某个位置的服务器）。在列表上下文中，`getgrent`的返回值是：

```
# 0      1      2      3
($name, $passwd, $gid, $members) = getgrent();
```

其中`$members`包含一个组成员登录名列表，各个登录名之间用空格分隔。要建立一个散列将组名转换为GID，可以写为：

```
while (($name, $passwd, $gid) = getgrent()) {
    $gid{$name} = $gid;
}
```

在标量上下文中，`getgrent`只返回组名。对于这个函数，标准`User::grent`模块支持一个按名访问的接口。另外参见`getgrent(3)`。

## getgrgid



```
getgrgid GID
```

这个函数按组号查找一个组文件记录。在列表上下文中，其返回值为：

```
# 0      1      2      3
($name, $passwd, $gid, $members)
    = getgrgid(0);
```

其中`$members`包含一个组成员登录名列表，各登录名之间用空格分隔。如果要反复这样做，可以考虑使用`getgrent`把数据缓存在一个散列中。

在标量上下文中，`getgrgid`只返回组名。对于这个函数，标准`User::grent`模块支持一个按名访问的接口。另外参见`getgrgid(3)`。

## getgrnam



```
getgrnam NAME
```

这个函数按组名查找一个组文件记录。在列表上下文中，其返回值为：

```
# 0      1      2      3
($name, $passwd, $gid, $members) =
    getgrnam("wheel");
```

其中`$members`包含一个组成员登录名列表，各登录名之间用空格分隔。如果要反复这样做，可以考虑使用`getgrent`把数据缓存在一个散列中。

在标量上下文中，`getgrnam`只返回数值组ID。对于这个函数，标准`User::grent`模块支持一个按名访问的接口。另外参见`getgrnam(3)`。

## gethostbyaddr



```
gethostbyaddr ADDR, ADDRTYPE
```

这个函数将地址转换为名字（和候选地址）。`ADDR`应当是一个打包的二进制网络地址，`ADDRTYPE`实际上总是`AF_INET`（来自`Socket`模块）。在列表上下文中，其返回值为：

```
# 0      1      2      3      4 ...
($name, $aliases, $addrtype, $length, @addrs) =
    gethostbyaddr($packed_binary_address, $addrtype);
```

这里`@addrs`是一个打包二进制地址列表。在Internet网域中，每个地址（历史上）都为4个字节长，可以用以下代码解包：

```
($a, $b, $c, $d) = unpack("C4", $addrs[0]);
```

或者，可以对`sprintf`使用`v`修饰符直接转换为点向量记法：

```
$dots = sprintf "%vd", $addrs[0];
```

`Socket`模块的`inet_ntoa`函数对于生成可打印的版本很有用。如果我们都打算转向IPv6，这个方法会变得很重要。

```
use Socket;
$printable_address = inet_ntoa($addrs[0]);
```

在标量上下文中，`gethostbyaddr`只返回主机名。

要从一个点向量生成一个`ADDR`，可以写为：

```
use Socket;
$ipaddr = inet_aton("127.0.0.1"); # 本地主机
$sclaimed_hostname = gethostbyaddr($ipaddr, AF_INET);
```

有关的更多例子参见第15章“套接字”一节。对于这个函数，`Net::hostent`模块提供了一个按名访问的接口。另外参见`gethostbyaddr(3)`。

`Socket`模块有一个`gethostinfo`函数，适用于所有常用形式的地址，包括IPv6。

`getaddrinfo`函数用于得到给定主机（以及可能的服务）的IP地址和端口号的一个列表，与`gethostbyname(3)`和`getservbyname(3)`函数本比，它提供了更大的灵活性。

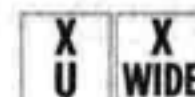
```

use Socket qw(:all);
@addr_structs = getaddrinfo("127.0.0.1"); # IPv4回送
@addr_structs = getaddrinfo("207.171.7.72");

@addr_structs = getaddrinfo("::1"); # IPv6回送
@addr_structs = getaddrinfo("e80::223:12ff:fe58:714c");

```

## gethostbyname



gethostbyname *NAME*

这个函数将一个网络主机名转换为相应的地址（及其他名字）。在列表上下文中，其返回值为：

```

# 0      1      2      3      4 ...
($name, $aliases, $addrtype, $length, @addrs) =
    gethostbyname($remote_hostname);

```

这里@addrs是一个原始地址列表。在Internet网域中，每个地址（在历史上）为4个字节长，可以如下解包：

```
($a, $b, $c, $d) = unpack("C4", $addrs[0]);
```

可以对sprintf使用v修饰符直接转换为点向量记法：

```
$dots = sprintf "%vd", $addrs[0];
```

在标量上下文中，gethostbyname只返回主机地址：

```

use Socket;
$ipaddr = gethostbyname($remote_host);
printf "%s has address %s\n",
    $remote_host, inet_ntoa($ipaddr);

```

另一个方法参见第15章“套接字”一节。对于这个函数，Net::hostent模块提供了一个按名访问的接口。另外参见gethostbyname(3)。

## gethostent



```

gethostent
sethostent STAYOPEN
endhostent

```

这些函数会迭代处理你的/etc/hosts文件，一次返回一条记录。gethostent的返回值为：

```
($name, $aliases, $addrtype, $length, @addrs)
```

这里@addrs是一个原始地址列表。在Internet网域中，每个地址（在历史上）为4个字节长，可以如下解包：



```
($a, $b, $c, $d) = unpack("C4", $addrs[0]);
```

一般认为使用`gethostent`的脚本不是可移植的。如果一个机器使用某个名字服务器，对于要得到全球每一台主机地址的请求，为了满足这个请求，它必须查询Internet的大部分地址。所以这些机器上没有实现`gethostent`。另外参见`gethostent(3)`了解其他详细信息。

对于这个函数，`Net::hostent`模块提供了一个按名访问的接口。

## getlogin



`getlogin`

这个函数返回当前登录名（如果能找到）。在Unix系统上，这要从`utmp(5)`文件读取。如果返回`false`，则使用`getpwuid`。例如：

```
$login = getlogin() || (getpwuid($<))[0] || "Intruder!!";
```

## getnetbyaddr



`getnetbyaddr ADDR, ADDRTYPE`

这个函数将一个网络地址转换为相应的网络名。在列表上下文中，其返回值为：

```
use Socket;
($name, $aliases, $addrtype, $net) = getnetbyaddr(127, AF_INET);
```

在标量上下文中，`getnetbyaddr`只返回网络名。对于这个函数，`Net::netent`模块提供了一个按名访问的接口。另外参见`getnetbyaddr(3)`。

## getnetbyname



`getnetbyname NAME`

这个函数将一个网络名转换为相应的网络地址。在列表上下文中，其返回值为：

```
($name, $aliases, $addrtype, $net) = getnetbyname("loopback");
```

在标量上下文中，`getnetbyname`只返回网络地址。对于这个函数，`Net::netent`模块提供了一个按名访问的接口。另外参见`getnetbyname(3)`。

## getnetent



```
getnetent
setnetent STAYOPEN
endnetent
```

这些函数迭代处理你的`/etc/networks`文件。在列表上下文中，其返回值为：

```
($name, $aliases, $addrtype, $net) = getnetent();
```

在标量上下文中，`getnetent`只返回网络名。对于这个函数，`Net::netent`模块提供了一个按名访问的接口。另外参见`getnetent(3)`。

如今，网络名的概念看起来很奇怪，大多数IP地址都在无名的（或不可命名的）子网上。

## getpeername

\$!	X	X
	ARG	U

`getpeername SOCKET`

这个函数返回连接`SOCKET`另一端的打包套接字地址。例如：

```
use Socket;
$hersockaddr      = getpeername SOCK;
($port, $heraddr) = sockaddr_in($hersockaddr);
$herhostname      = gethostbyaddr($heraddr, AF_INET);
$herstraddr       = inet_ntoa($heraddr);
```

## getpgrp

\$!	X
	U

`getpgrp PID`

这个函数返回指定`PID`的当前进程组（使用`PID 0`表示当前进程）。在一个没有实现`getpgrp(2)`的机器上，如果调用`getpgrp`会产生一个异常。如果省略`PID`，函数会返回当前进程的进程组（等同于使用`PID`为0）。在用POSIX `getpgrp(2)`系统调用实现这个操作符的系统上，必须省略`PID`，即使要提供也必须为0。

## getppid

X
U

`getppid`

这个函数返回父进程的进程ID。在典型的Unix系统上，如果父进程ID改为1，表示父进程已经结束，你已经被`init(8)`程序接收。

## getpriority

\$!	X
	U

`getpriority WHICH, WHO`

这个函数返回一个进程、进程组或用户的当前优先级。参见`getpriority(2)`。在一个没有实现`getpriority(2)`的机器上，如果调用`getpriority`会产生一个异常。CPAN上的`BSD::Resource`模块提供了一个更方便的接口，包括`PRIO_PROCESS`、`PRIO_PGRP`和`PRIO_USER`符号常量，可以用来提供`WHICH`参数。尽管这些符号常量通常分别设置为0、1和2，不过你并不知道在C的`#include`文件的黑暗领地里会发生什么。

如果`WHO`值为0，表示当前进程、进程组或用户。所以要得到当前进程的优先级，可以使用：

```
$curprio = getpriority(0, 0);
```

## getprotobyname



getprotobyname *NAME*

这个函数将一个协议名转换为相应的协议号。在列表上下文中，其返回值为：

```
($name, $aliases, $protocol_number) = getprotobyname("tcp");
```

在标量上下文中调用时，`getprotobyname`只返回协议号。

对于这个函数，`Net::proto`模块提供了一个按名访问的接口。另外参见 `getprotobyname(3)`。

## getprotobynumber



getprotobynumber *NUMBER*

这个函数将一个协议号转换为相应的协议名。在列表上下文中，其返回值为：

```
# 1      2      3
($name, $aliases, $protocol_number) = getprotobynumber(6);
```

在标量上下文中调用时，`getprotobynumber`只返回协议名。对于这个函数，`Net::proto`模块提供了一个按名访问的接口。另外参见 `getprotobynumber(3)`。

## getprotoent



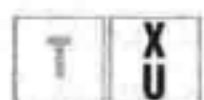
```
getprotoent
setprotoent STAYOPEN
endprotoent
```

这些函数迭代处理`/etc/protocols`文件。在列表上下文中，`getprotoent`的返回值为：

```
# 1      2      3
($name, $aliases, $protocol_number) = getprotoent();
```

在标量上下文中调用时，`getprotoent`只返回协议名。对于这个函数，`Net::proto`模块提供了一个按名访问的接口。另外参见 `getprotent(3)`。

## getpwent



getpwent



```
setpwent
endpwent
```

从概念上讲，这些函数会迭代处理`/etc/passwd`文件，不过如果你是超级用户，而且使用了shadow口令，这可能还涉及`/etc/shadow`文件。如果你在使用某个基于数据库或基于网络的登录系统，这会更为复杂。在列表上下文中，其返回值为：

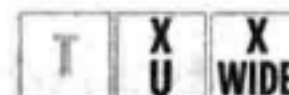
```
# 0      1      2      3      4      5      6      7      8
($name,$passwd,$uid,$gid,$quota,$comment,$gcos,$dir,$shell) = getpwent();
```

有些机器可能会把配额（quota）和注释（comment）字段用于其他用途（而不是其名字原来所指的用途），不过其余的字段是一样的。要建立一个散列，将登录名转换为UID，可以写为：

```
while (($name, $passwd, $uid) = getpwent()) {
    $uid{$name} = $uid;
}
```

在标量上下文中，`getpwent`只返回用户名。对于这个函数，`User::pwent`模块提供了一个按名访问的接口。另外参见`getpwent(3)`。

## getpwnam



```
getpwnam NAME
```

这个函数将一个用户名转换为相应的`/etc/passwd`文件记录。在列表上下文中，其返回值为：

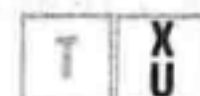
```
# 0      1      2      3      4      5      6      7      8
($name,$passwd,$uid,$gid,$quota,$comment,$gcos,$dir,$shell) = getpwnam("daemon");
```

在支持shadow口令的系统上，你必须是超级用户才能获取具体的口令。C函数库应当注意到你有适当的权限，并打开`/etc/shadow`文件（或者保存shadow口令的某个文件）。至少应该是这样。

对于重复的查找，可以考虑使用`getpwent`将数据缓存在一个散列中。

在标量上下文中，`getpwnam`只返回数值用户ID。对于这个函数，`User::pwent`模块提供了一个按名访问的接口。另外参见`getpwnam(3)`和`passwd(5)`。

## getpwuid



```
getpwuid UID
```

这个函数将一个数值用户ID转换为相应的`/etc/passwd`文件记录。在列表上下文中，其返回值为：

```
# 0      1      2      3      4      5      6      7      8
($name,$passwd,$uid,$gid,$quota,$comment,$gcos,$dir,$shell) = getpwuid(2);
```

对于重复的查找，可以考虑使用`getpwent`将数据缓存在一个散列中。

在标量上下文中，`getpwuid`返回用户名。对于这个函数，`User::pwent`模块提供了一个按名访问的接口。另外参见`getpwnam(3)`和`passwd(5)`。

## getservbyname



```
getservbyname NAME, PROTO
```

这个函数将一个服务（端口）名转换为相应的端口号。*PROTO*是一个协议名，如“tcp”。在列表上下文中，其返回值为：

```
# 0      1      2      3
($name, $aliases, $port_number, $protocol_name) = getservbyname("www", "tcp");
```

在标量上下文中，`getservbyname`只返回服务端口号。对于这个函数，`Net::servent`模块提供了一个按名访问的接口。另外参见`getservbyname(3)`。

## getservbyport



```
getservbyport PORT, PROTO
```

这个函数将一个服务（端口）号转换为相应的端口名。*PROTO*是一个协议名，如“tcp”。在列表上下文中，其返回值为：

```
# 0      1      2      3
($name, $aliases, $port_number, $protocol_name) = getservbyport(80, "tcp");
```

在标量上下文中，`getservbyport`只返回服务名。对于这个函数，`Net::servent`模块提供了一个按名访问的接口。另外参见`getservbyport(3)`。

## getservent



```
getservent
setservent STAYOPEN
endservent
```

这个函数迭代处理`/etc/services`文件或相应文件。在列表上下文中，其返回值为：

```
# 0      1      2      3
($name, $aliases, $port_number, $protocol_name) = getservent();
```

在标量上下文中，`getservent`只返回服务端口名。对于这个函数，`Net::servent`模块提供了一个按名访问的接口。另外参见`getservent(3)`。

## getsockname

\$!	X	X
	ARG	U

getsockname *SOCKET*

这个函数返回*SOCKET*连接这一端的打包套接字地址（为什么会不知道你自己的地址？可能是因为在完成一个accept之前，你绑定了一个包含通配符的服务器套接字地址，现在需要知道某人使用什么接口与你连接，或者父进程向你传入了一个套接字，如inetd）。

```
use Socket;
# 假设SOCK是一个已连接的套接字
$mysockaddr = getsockname(SOCK);
($port, $myaddr) = sockaddr_in($mysockaddr);
$myname = gethostbyaddr($myaddr, AF_INET);
printf "I am %s [%vd]\n", $myname, $myaddr;
```

## getsockopt

\$!	X	X
	ARG	U

getsockopt *SOCKET, LEVEL, OPTNAME*

这个函数查询给定*LEVEL*上与*SOCKET*关联的名为*OPTNAME*的选项。选项可能存在于多个协议层次上，这取决于套接字类型，不过至少会存在于最顶层套接字层SOL\_SOCKET（在Socket模块中定义）。要查询另一层上的选项，应当提供控制选项的适当协议的协议号。例如，要指示一个选项由TCP协议解释，*LEVEL*应当设置为TCP的协议号，这可以使用getprotobyname得到。

这个函数返回一个打包的字符串，表示所请求的套接字选项，如果由于\$!中的错误而导致出错，会返回undef。打包字符串中的内容取决于*LEVEL*和*OPTNAME*，参见getsockopt(2)了解详细信息。不过通常选项是一个整数，在这种情况下结果就是一个打包的整数，可以用“i”（或I）格式使用unpack解包。

例如，要测试套接字上是否启用了Nagle算法：

```
use Socket qw(:all);

# 假设$socket包含一个已连接套接字的句柄
$tcp = IPPROTO_TCP;
$packed = getsockopt($socket, $tcp, TCP_NODELAY)
    || die "getsockopt: $!";
$nodelay = unpack("I", $packed);
printf "Nagle's algorithm is o%s.\n", $nodelay ? "ff" : "n";
```

更多信息请参见setsockopt。

## glob

\$_	\$@	T	X
		T	T

```
glob EXPR
glob
```



这个函数返回`EXPR`的值，其中包含的文件扩展名类似于shell指定的文件扩展名。这是实现`<*>`操作符的内部函数。

由于历史原因，算法匹配`cs(1)`风格而不是Bourne shell风格的扩展名。首字符为点号（“.”）的文件会被忽略，除非这个字符显式匹配。星号（“\*”）匹配任意字符（包括无字符）的任意序列。问号（“?”）匹配任意单个字符。中括号序列（“[...]”）指定一个简单字符类，如“[chy0-9]”。字符类可以用一个脱字符取反，如“\*.[^oa]”，这会匹配扩展名包括一个点号后面跟有一个字符的所有文件，扩展名中最后的这个字符不能是“a”或“o”。波浪线（“~”）扩展到主目录，如“~/.`*rc`”表示所有当前用户的“rc”文件，“~jane/Mail/`*`”表示Jane的所有邮件文件。大括号可以用于完成替换，如“~/.{mail,ex,cs,tm,}rc”可以获取那些特定的rc文件。

`glob`函数仍用老办法，使用空白符来分隔多个模式，如`<*.c *.h>`。如果你想对可能包含空白符的文件名聚团，必须使用另外的引号包围有空格的文件名来提供保护。例如，要对包含一个“e”后面跟有一个空格再后面是一个“f”的文件名聚团，可以使用以下代码：

```
@spaces = <"*e f*>;
@spaces = glob '"*e f*"' ;
@spaces = glob q(*e f*) ;
```

如果必须传入一个变量，可以这样做：

```
@spaces = glob "'*${var}e f*'" ;
@spaces = glob qq(*${var}e f*) ;
```

或者，可以直接使用`File::Glob`模块；有关的详细信息参见这个模块的手册页。调用`glob`或`<*>`操作符会自动用`use`加载该模块，所以如果模块神秘地从库中消失了，会产生一个异常。

调用`open`时，Perl不会扩展通配符，甚至不会扩展波浪线。需要首先用`glob`处理结果：

```
open(MAILRC, "~/.mailrc")           # 不正确：波浪线是shell符号
|| die "can't open ~/.mailrc: $!";

open(MAILRC, <~/.mailrc>)           # 先扩展波浪线
|| die "can't open ~/.mailrc: $!";

open(MAILRC, (glob("~/.mailrc"))[0]) # 同样，但更多
|| die "can't open ~/.mailrc: $!";   # 当心列表返回
```

如果非空大括号是`glob`中使用的唯一的通配符，这不会匹配任何文件名，不过可能会返回很多字符串。例如，这会生成9个字符串，分别对应一组水果和颜色：

```
@many = glob "{apple,tomato,cherry}={green,yellow,red}";
```

`glob`函数与Perl的类型团概念没有任何关系，只不过它们都使用一个`*`表示多个项。参见第2章中“文件名聚团操作符”一节。

## gmtime

```
gmtime EXPR
gmtime
```

这个函数将`time`函数返回的一个时间转换为与格林威治时间校准的时间，返回一个包含9个元素的列表。格林威治时间历史上称为Greenwich Mean Time (GMT)，不过现在称为世界统一时间 (Coordinated Universal Time, UTC)。通常如下使用：

```
# 0 1 2 3 4 5 6 7 8
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = gmtime;
```

如果类似这样省略`EXPR`，则完成`gmtime(time())`。Perl库模块`Time::Local`包含一个子例程`timegm`，它可以将列表转换回一个时间值。

所有列表元素都是数值，直接取自一个`struct tm`（别害怕，这是一个C编程结构）。具体地，这表示`$mon`范围为0..11，1月作为月份0，`$wday`范围为0..6，星期天作为日期0。很容易记住哪些是基于0的，因为在包含月份和日期名的基于0的数组中，通常就是使用这些作为下标。

例如，要得到伦敦的当前月份，可以写为：

```
$london_month = (qw(Jan Feb Mar Apr May Jun
                    Jul Aug Sep Oct Nov Dec))[(gmtime)[4]];
```

`$year`是自1900年以后的年数。也就是说，如果是2023年，`$year`就是123，而不是23。要得到4位的年份，应该写为`$year + 1900`。要得到两位的年份（例如，2001年应该是“01”），可以使用`sprintf("%02d", $year % 100)`。

在标量上下文中，`gmtime`根据GMT时间值返回一个类`ctime(3)`的字符串。`Time::gmtime`模块为这个函数提供了一个按名访问的接口。另外参见`POSIX::strftime`来了解格式化时间的一个更细粒度的方法。

这个标量值并不依赖于本地化环境，这是一个Perl内置的标量。同样参见`Time::Local`模块，还可以参见通过`POSIX`模块得到的`strftime(3)`和`mktime(3)`函数。要得到类似但依赖于本地化环境的日期字符串，需要适当地建立你的本地化环境变量（请参见`perllocale`手册页），并尝试以下代码：

```
use POSIX qw(strftime);
$now_string = strftime "%a %b %e %H:%M:%S %Y", gmtime;
```

可能并不是所有本地化环境中`%a`和`%b`转义（分别表示星期几和几月的短形式）都是3个字符宽。



## goto

```
goto LABEL
goto EXPR
goto &NAME
```

`goto LABEL`找到标有`LABEL`的语句，并从那里继续执行。如果没有找到`LABEL`，会产生一个异常。不能用这个函数进入任何需要初始化的构造，如子例程或`foreach`循环。也不能用来进入已经优化的构造。但是`goto`可以进入动态作用域中几乎所有其他地方<sup>注5</sup>，包括例程之外。不过，对此通常更好的办法是使用另外某个构造，如`last`或`die`。Perl的作者从来都不认为有必要使用这种形式的`goto`（在Perl中是这样，不过C中则是另一回事）。

更复杂（甚至更愚蠢）的是，Perl还允许`goto EXPR`，这里要求`EXPR`计算一个标签名，在运行时之前其位置肯定不可解析，因为编译这个语句时标签是未知的。这样可以支持类似`FORTRAN`的计算型`goto`，不过如果你要优化来提高可维护性，则不建议这么做：

```
goto +("FOO", "BAR", "GLARCH")[$i];
```

不相关的`goto &NAME`非常神奇，它能用命名子例程调用替换当前运行的子例程。可以使用这个构造，而不像使用`AUTOLOAD`子例程那么麻烦，因为使用`AUTOLOAD`会加载另一个子例程，然后假装一开始调用的就是这个新子例程（而不是原来的子例程），除非原子例程中对`@_`的修改已经传播到作为替换的新子例程中。在`goto`之后，甚至`caller`都无法区分出最初调用的是原来的`AUTOLOAD`例程。

## grep

```
grep EXPR, LIST
grep BLOCK LIST
```

这个函数在布尔上下文为`LIST`的各个元素计算`EXPR`或`BLOCK`，依次将`$_`临时设置为各个元素，这与`foreach`构造很类似。在列表上下文中，它返回一个列表，其中包含`LIST`中该表达式计算为`true`的元素（这个操作符是根据大家喜爱的一个Unix程序来命名的，它能从文件中抽取出与特定模式匹配的行。在Perl中，表达式通常是一个模式，不过也不一定）。在标量上下文中，`grep`返回表达式计算为`true`的次数。

如果`@all_lines`包含代码行，下面这个例子会剔除注释行：

```
@code_lines = grep !/^\\s*#/, @all_lines;
```

由于`$_`是各个列表值的一个隐式别名，修改`$_`也会修改原列表中的元素。尽管这是允许的，也很有用，但有时如果你没有心理准备，可能得到奇怪的结果。例如：

注5：这说明，如果在当前例程中没有找到这个标签，它会回溯查找调用当前例程的例程，来寻找这个标签，这会使你的程序几乎无法维护。



```
@list = qw(barney fred dino wilma);
@greplist = grep { s/^[bfd]// } @list;
```

@greplist现在是“arney”，“red”，“ino”，而@list现在是“arney”，“red”，“ino”，“wilma”！所以，一定要当心。

参见map。下面两个语句的作用是等价的：

```
@out = grep { EXPR } @in;
@out = map { EXPR ? $_ : () } @in
```

对于短路版本的grep，参见标准List::Util模块的first函数。它并不是返回EXPR计算为true的所有元素的列表，而只返回使EXPR为true的第一个元素，或者如果没有这样的元素，则返回undef。与以往一样，\$\_会分别设置为各个元素：

```
use List::Util qw(first);

$first_over_100 = first { $_ > 100 } @list;
$first_with_foo = first { /foo/ } @list;
```

下面这个函数取一个字符，并报告它首次出现在哪个版本的Unicode标准中：

```
use v5.14;
use List::Util qw(first);
sub getage(_) {
    my $one_char = shift;
    die unless length($one_char) == 1;
    state $ages = [reverse qw(1.1 2.0 2.1 3.0 3.1 3.2
                               4.0 4.1 5.0 5.1 5.2 6.0
                               )];
    return first { $one_char =~ /\p{Age=$_}/ } @$ages;
}
```

## hex

\$\_

```
hex EXPR
hex
```

这个函数将EXPR解释为一个十六进制字符串，并返回等价的十进制值。如果EXPR有前导“0x”，则将被忽略。如果要解释可能以0、0b或0x开头的字符串，参见oct。下面的代码会把\$number设置为4294906560：

```
$number = hex("ffff12c0");
```

要完成逆向功能，可以使用sprintf：

```
sprintf "%lx", $number;      # (那是一个字母l，而不是数字1)
```

十六进制串只能表示整数。导致整数溢出的字符串会触发一个警告。不同于oct，这个函数不会去除前导空白符。

## import

```
import CLASSNAME LIST
import CLASSNAME
```

没有内置的import函数。这只是模块定义（或继承）的一个普通的类方法，希望通过use操作符向另一个模块导出名字。详细内容参见use。

## index

```
index STR, SUBSTR, OFFSET
index STR, SUBSTR
```

这个函数在另一个字符串中搜索一个直接量字符串。它会返回STR中SUBSTR第一次出现的位置。如果指定了OFFSET，这个参数指出了开始查找之前需要从头跳过多少个字符。这个位置是基于0的。如果未找到子串，函数会返回基数减1，通常是-1。要遍历一个字符串，可以写为：

```
$pos = -1;
while (($pos = index($string, $lookfor, $pos)) > -1) {
    say "Found at $pos"; $pos++;
}
```

偏移量总是按程序员可见的字符来确定（即码点），而不是按用户可见的字符（即字形）来确定。只有当已经从抽象字符解码为某种串行化机制时，偏移量才以字节为单位，如UTF-8或UTF-16。参见第6章。

要处理作为字形序列而不是码点序列的字符串，参见CPAN Unicode::GCString模块的index、rindex和pos方法。

## int

\$\_

```
int EXPR
int
```

这个函数返回EXPR的整数部分。如果你是一个C程序员，通常可能会忘记对除法使用int，在Perl中除法是一个浮点数操作：

```
$average_age = 939/16;      # 得到58.6875（在C中会得到58）
$average_age = int 939/16;  # 得到58
```

不能使用这个函数作为取整方法，因为它会向0截断，另外浮点数的机器表示可能会生成不太直观的结果。例如，int(-6.725/0.025)会生成-268而不是正确的-269；这是因为这个值实际上更像-268.99999999999994315658。通常情况下，sprintf、printf或POSIX::floor以及POSIX::ceil函数都比int好。

```
$n = sprintf("%.0f", $f);    # 取整（而不是截断）为最接近的整数
```

为了补偿“四舍五入”可能带来的固有偏差，IEEE指出，对于5要取整为最接近的偶数。因此，以下代码：

```
for (-3 ... 3) { printf "%.0f\n", $_ + 0.5 }
```

会打印序列-2, -2, -0, 0, 2, 2和4。

## ioctl



```
ioctl FILEHANDLE, FUNCTION, SCALAR
```

这个函数实现了控制I/O的*ioctl(2)*系统调用。为了得到正确的函数定义，首先可能必须声明：

```
require "sys/ioctl.ph"; # 可能是/usr/local/lib/perl/sys/ioctl.ph
```

如果*sys/ioctl.ph*不存在或者没有正确的定义，必须根据你的C头文件（如*sys/ioctl.h*）建立你自己的*sys/ioctl.ph*（Perl发行版包括一个名为*h2ph*的脚本可能会有帮助，不过运行这个脚本不是一件轻松的事情）。将根据*FUNCTION*读或写（或者读写）*SCALAR*，*SCALAR* 字符串值的一个指针将作为实际*ioctl(2)*调用的第3个参数传入。如果*SCALAR*没有字符串值，而是有一个数字值，将直接传递这个值，而不是传递字符串值的一个指针。*pack*和*unpack*函数对于管理*ioctl*使用的结构值很有用。如果*ioctl*需要向*SCALAR*写数据，你要确保这个字符串足够长，可以容纳需要写的全部数据，为此通常可以使用*pack*将它初始化为适当的大小。下面的例子会确定用FIONREAD *ioctl*可以读取多少个字节（而非字符）：

```
require "sys/ioctl.ph";

# 预分配适当大小的缓冲区：
$size = pack("L", 0);
ioctl(FH, FIONREAD(), $size)
    || die "Couldn't call ioctl: $!";
$size = unpack("L", $size);
```

下面展示了如何检测当前窗口大小<sup>注6</sup>（行列数）：

```
require "sys/ioctl.ph";

# 原生大小为4个无符号short
$template = "S!4";
# 预分配适当大小的缓冲区：
my $ws = pack($template, ());
ioctl(STDOUT, TIOCGWINSZ(), $ws)
    || die "Couldn't call ioctl: $!";
```

---

注6： 或者如何得到与STDOUT文件句柄关联的窗口大小。



```
($rows, $cols, $xpix, $ypix) = unpack($template, $ws);
```

如果`h2ph`未安装，或者不可用，可以使用`grep`手动地搜索包含文件，或者写一个小C程序来打印这个值。还要查看C代码来确定系统所需的结构模板布局 and 大小。

`ioctl`（和`fcntl`）的返回值如表27-2所示。

表27-2: `ioctl`的返回值

系统调用返回值	Perl返回值
-1	undef
0	字符串 "0 but true"
任何其他值	该数字

因此，成功时Perl返回`true`，失败时返回`false`，不过仍然可以很容易地确定操作系统返回的实际值：

```
$retval = ioctl(...) || -1;
printf "ioctl actually returned %d\n", $retval;
```

特殊字符串 "0 but true" 不会收到`-w`命令行标志或`warnings pragma`关于不正确的数值转换发出的警告。

不能认为`ioctl`调用是可移植的。例如，如果只想对整个脚本关闭一次回显，更可移植的做法是：

```
system "stty -echo"; # 适用于大多数Unix机器
```

只是因为你在Perl中可以做某件事并不意味着你应该那么做。为了得到更好的可移植性，可以查看CPAN的`Term::ReadKey`模块。你想用`ioctl`完成的几乎每一个工作可能都已经有了一个相应的CPAN模块，而且采用了更可移植的方式，因为它们会利用系统的C编译器来完成繁重的具体事务。

## join

```
join EXPR, LIST
```

这个函数将`LIST`中单个的字符串连接为一个字符串，各个字段用`EXPR`的值分隔，并返回这个字符串。例如：

```
$rec = join ":", $login,$passwd,$uid,$gid,$gcos,$home,$shell;
```

如果要反过来，参见`split`。要按固定位置的字段连接起来，参见`pack`。如果要连接很多个字符串，最高效的方法是利用一个`null`字符串使用`join`来连接：

```
$string = join "", @array;
```

与`split`不同，`join`不是以模式作为它的第一个参数，如果你打算这么做，会生成一个警告。

## keys



```
Keys HASH  
keys ARRAY  
keys EXPR
```

这个函数返回一个列表，其中包含指定散列中的所有键。这些键以一种看上去随机的顺序返回，不过这个顺序与`values`或`each`函数生成的顺序相同（假设这个散列在调用之间未被修改）。作为一个副作用，它会重置`HASH`的迭代器。下面的方法（相当笨）可以用来打印你的环境：

```
@keys = keys %ENV;      # keys与  
@values = values %ENV; # values的顺序相同，如这里所示  
while (@keys) {  
    say pop(@keys), "=", pop(@values);  
}
```

你更希望看到按键排序的环境：

```
for my $key (sort keys %ENV) {  
    say $key, "=", $ENV{$key};  
}
```

可以直接对散列的值排序，不过如果没有办法把值映射回键，这没有太大用处。要按值对一个散列排序，通常需要提供一個根据键访问值的比较函数来对`keys`排序。下面会按值将一个散列以降序顺序排序：

```
for my $key (sort { $hash{$b} <=> $hash{$a} } keys %hash) {  
    printf "%4d %s\n", $hash{$key}, $key;  
}
```

如果散列绑定到一个相当大的DBM文件，对这样一个散列使用`keys`时，会生成一个相当大的列表，这会导致一个相当大的进程。在这种情况下，你可能更愿意使用`each`函数，它会逐个地迭代处理散列记录，而不是将它们通通吞到一个超大的列表中。

在标量上下文中，`keys`返回散列的元素个数（并重置`each`迭代器）。不过，要得到绑定散列的这个信息，包括DBM文件，Perl必须遍历整个散列，所以不够高效。在`void`上下文调用`keys`会有帮助。

用作为一个左值时，`keys`会增加为给定散列分配的散列桶数（这类似于为`$#array`赋一个更大的数来预扩展数组）。如果你知道散列会增长，而且知道它会变到多大，预扩展散列可以得到一定的效率提升。如果有：

```
keys %hash = 1000;
```

那么%hash就会至少分配1000个桶（实际上，你会得到1024个桶，因为它会向上取整为下一个2的幂）。不能使用keys以这种方式缩减为散列分配的桶数（不过即使不小心这样做了也不用担心，因为这样没有任何效果）。即使有%hash = (), 桶也会保留。如果%hash仍在作用域中而你想释放存储空间，可以使用undef %hash。

参见each、values和sort。

## kill



```
kill SIGNAL, LIST
```

这个函数向一个进程列表发出一个信号。对于SIGNAL，可以使用一个整数，或者一个加引号的信号名（前面没有“SIG”）。如果试图使用一个无法识别的SIGNAL名，会产生一个异常。这个函数会返回成功得到信号的进程数。如果SIGNAL为负值，这个函数会关闭进程组而不是进程（在由SysV发展而来的Unix系统上，负进程号也会关闭进程组，不过这是不可移植的）。PID为0时，会把信号发送到与发送者有相同组ID的所有进程。

例如：

```
$cnt = kill 1, $child1, $child2;
kill 9, @goners;
kill "STOP", getppid      # 可能*会*挂起登录shell...
                        unless getppid == 1; # 但不影响init(8)
```

SIGNAL为0时，会测试一个进程是否仍存活，另外你是否有权向它发出信号。这里并不会真正发送任何信号。这样一来，你可以检查进程是否存在，而且并没有改变它的UID。

```
use Errno qw(ESRCH EPERM);
if (kill 0 => $minion) {
    say "$minion is alive!";
} elsif ($! == EPERM) { # 改变了UID
    say "$minion has escaped my control!";
} elsif ($! == ESRCH) {
    say "$minion is deceased."; #或变成僵尸进程
} else {
    warn "Odd; I couldn't check on the status of $minion: $!\n";
}
```

参见第15章“信号”一节。

## last



```
last LABEL
```



last

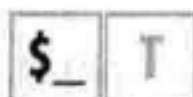
last操作符会立即退出当前循环，就像C或Java中（用于循环中）的break语句。如果省略LABEL，这个操作符则指示最内层包围循环。如果有continue块，将不会执行。

```
LINE: while (<MAILMSG>) {  
    last LINE if /^$/; # 处理完首部后退出  
    # 循环的其余部分  
}
```

不能用last退出一个有返回值的块，如eval {}、sub{}或do {}，另外不要用它退出一个grep或map操作。如果启用了警告，使用last退出一个不在当前词法作用域中的循环时Perl会发出警告，如一个调用子例程中的循环。

块本身在语义上等同于只执行一次的循环。因此，last可以用于提前退出这样一个块。另外参见第4章对last、next、redo和continue的介绍。

## lc



```
lc EXPR  
lc
```

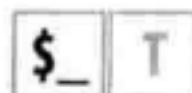
这个函数返回EXPR的一个小写版本。这是实现双引号字符串中\L转义的内部函数。

不要使用lc来完成不区分大小写的比较，在ASCII中你可能可以这么做，但是对于Unicode这会给出错误的答案。实际上，应当使用fc (foldcase) 函数，可以利用CPAN Unicode::CaseFold模块，或者通过v5.16或以后版本中新增的use feature "fc"。另外参见第6章“张冠李戴”一节了解更多信息。

如果字符串没有Unicode语义（而且本地化环境模式未生效），128~256范围内的码点会被lc忽略，这一点可能很难猜测。unicode\_strings特性可以确保这些码点也有Unicode语义。参见第6章。

如果use locale生效，会参考你当前的LC\_CTYPE本地化环境，不过据说关于本地化环境与Unicode如何交互还在研究当中。另外参见perllocale、perlunicode和Lperlfunc>手册页了解最新结果。

## lcfirst



```
lcfirst EXPR  
lcfirst
```

这个函数返回首字母小写的EXPR版本。这是实现双引号字符串中\l转义的内部函数。关于Unicode大小写映射，参见前一项lc的说明。

## length

\$ <sub>-</sub>
-----------------

```
length EXPR
length
```

这个函数返回标量值*EXPR*的码点（程序员可见的字符）长度。如果省略*EXPR*，则返回\$<sub>-</sub>的长度（不过要当心，下一项不能看上去像一个*EXPR*的开头，否则Perl的词法分析器会不知所措。例如，`length < 10`就无法编译。如果有二义性，应当使用小括号）。

不要试图使用`length`来得到一个数组或散列的大小。要得到一个数组的大小，应当使用`scalar @array`，要得到散列中键/值对的个数，应当使用`scalar keys %hash`（`scalar`多余时通常会省略）。

要得到一个字符串中的字形（用户可见的字符）个数，可以直接统计：

```
my $count = 0;
$count++ while our $string =~ /\X/g;
```

也可以使用CPAN `Unicode::GCString`模块来得到字形个数，它允许把字符串处理为一个字形序列而不是一个码点序列。这个模块还可以告诉你一个字符串按打印列统计的长度。因此你仍然可以使用`printf`对齐，如果富有创造性，甚至还可以使用`format`和`write`（尽管打印时有些码点占0列，另外一些占1列，还有一些占2列）。

## \_\_LINE\_\_

这是一个特殊的token，会编译为当前行号。参见第21章“用其他语言生成Perl”一节。

## link

\$!	X T	X U
-----	--------	--------

```
link OLDFILE, NEWFILE
```

这个函数创建链接到原文件名的一个新文件名。成功时这个函数返回`true`，否则返回`false`。另外参见本章后面的`symlink`。这个函数不太可能在非Unix风格的文件系统上实现。

## listen

\$!	X ARG	X U
-----	----------	--------

```
listen SOCKET, QUEUESIZE
```

这个函数告诉系统你要在这个*SOCKET*上接收连接，而且系统会对等待的连接排队，等待的连接数目由*QUEUESIZE*指定。想象你有很多未接电话，多达17个人在排队等待（真吓人）。成功时这个函数返回`true`，否则返回`false`。

```
use Socket;
```

```
listen(PROTOSOCK, SOMAXCONN)
|| die "cannot set listen queue on PROTOSOCK: $!";
```

参见accept。另外参见第15章“套接字”一节。还可以参见`listen(2)`。

## local

```
local EXPR
```

这个操作符并不创建一个局部变量，如果要创建局部变量，应该使用`my`才对。实际上，`local`只是局部化现有的变量。也就是说，它会让一个或多个全局变量在最内层外围块、`eval`或文件中有局部作用域值。如果列出了多个变量，这个列表必须放在小括号里，因为这个操作符绑定比逗号更紧密。列出的所有变量都必须是合法的左值。也就是说，能够为这些变量赋值，这可能包括数组或散列的单个元素。

这个操作符的工作是将指定变量的当前值保存在一个隐藏的堆栈中，退出这个块、子例程、`eval`或文件时再恢复这个值。执行`local`之后而且在退出这个作用域之前，所有子例程和所执行的格式都会看到局部的内部值而不是之前的外部值，因为这个变量仍是一个全局变量，尽管它有一个局部化的值。相应的技术术语是建立动态作用域（dynamic scoping）。参见第4章“作用域声明”一节。

如果需要，可以为`EXPR`赋值，这允许你在局部化变量时对变量完成初始化。如果没有给定初始化代码，所有标量都会初始化为`undef`，所有数组和散列都初始化为`()`。与正常的赋值一样，如果你对左边的变量使用小括号（或者如果变量是一个数组或散列），右边的表达式将在列表上下文中计算。否则，右边的表达式会在标量上下文中计算。

不管怎样，右边的表达式都会在局部化之前计算，不过初始化要在局部化之后发生，所以可以用非局部化值初始化一个局部化变量。例如，以下代码展示了如何临时修改一个全局数组：

```
if ($sw eq "-v") {
    # 用全局数组初始化局部数组
    local @ARGV = @ARGV;
    unshift(@ARGV, "echo");
    system @ARGV;
}
# 恢复@ARGV
```

还可以临时修改全局散列：

```
# 为%digits散列临时增加两个记录
if ($base12) {
    # 注意：我们并不认为这种做法效率高！
    local(%digits) = (%digits, T => 10, E => 11);
    parse_num();
}
```



可以使用`local`为数组和散列的单个元素提供临时值，即使它是词法作用域变量：

```
if ($protected) {
    local $SIG{INT} = "IGNORE";
    precious();      # 这个函数中没有中断
}                  # 恢复之前的处理器（如果有）
```

还可以对类型团使用`local`来创建局部文件句柄，而无需加载庞大的对象模块：

```
local *MOTD;          # 保护全局MOTD句柄
my $fh = do { local *FH }; # 创建新的间接文件句柄
```

你可能会看到，比较老的代码中，如果需要生成新的文件句柄，往往会使用局部化类型团，但如今一个简单的`my $fh`就足够了。这是因为，如果提供一个未定义的变量作为文件句柄参数提供给一个初始化文件句柄的函数（如作为`open`或`socket`的第一个参数），Perl会为你自动生成一个全新的文件句柄。

一般来讲，通常都会使用`my`而不是`local`，因为`local`并不具有大多数人所认为的“局部”含义。参见`my`。

还可以用`delete local EXPR`构造将数组或散列元素的删除局部化到当前块。

## localtime

```
localtime EXPR
localtime
```

这个函数将`time`返回的值转换为一个包含9个元素的列表，表示已按照本地时区校准的时间。通常如下使用：

```
# 0   1   2   3   4   5   6   7   8
($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) = localtime;
```

如果省略`EXPR`（如这里所示），会完成`localtime(time())`。

所有列表元素都是数值，直接取自一个`struct tm`（别害怕，这是一个C编程结构）。具体地，这表示`$mon`范围为0..11，1月作为月份0，`$wday`范围为0..6，星期天作为日期0。很容易记住哪些是基于0的，因为在包含月份和日期名的基于0的数组中，通常就是使用这些作为下标。

例如，要得到当前星期几的名字，可以写为：

```
$thisday = (Sun,Mon,Tue,Wed,Thu,Fri,Sat)[(localtime)[6]];
```

`$year`是自1900年以后的年数。也就是说，如果是2023年，`$year`就是123，而不是23。要得到4位的年份，应该写为`$year + 1900`。要得到两位的年份（例如，2001年应该是“01”），可以使用`sprintf("%02d", $year % 100)`。

Perl库模块Time::Local包含一个子例程timelocal，它可以完成反方向的转换。

在标量上下文中，localtime返回一个类ctime(3)的字符串。例如（几乎）可以用以下代码模拟date(1)命令<sup>注7</sup>：

```
Perl-le 'print scalar localtime()'
```

更细粒度的格式化时间的方法参见标准POSIX模块的strftime函数。Time::localtime模块为这个函数提供了一个按名访问的接口。

## lock

```
lock THING
```

lock函数在一个变量、子例程或THING引用的对象上放置一个锁，直到这个锁超出其作用域。为了保证向后兼容，只有当你的Perl版本编译时启用了线程而且已经声明了use Threads时这个函数才是内置函数。否则，Perl会认为这是一个用户自定义的函数。

## log

`$_` `$@`

```
log EXPR  
log
```

这个函数返回EXPR的自然对数（以e为底）。如果EXPR为负数，会产生一个异常。要得到其他底数的对数，可以使用基本代数来计算：一个数以N为底的对数等于这个数的自然对数除以N的自然对数。例如：

```
sub log10 {  
    my $n = shift;  
    return log($n)/log(10);  
}
```

对于log的逆运算，参见exp。

## lstat

`$_` `$!` `$?`

```
lstat EXPR  
lstat
```

这个函数与Perl的stat函数所做的工作是一样的（包括设置特殊的\_文件句柄），不过，如果文件名的最后一部分是一个符号链接，则会对这个符号链接本身执行stat，而不是处理这个符号链接所指示的文件。如果你的系统上未实现符号链接，则会完成一个普通的stat操作。

---

注7： date(1) 打印时区，而标量localtime不打印时区。

m//



```
/PATTERN/  
m/PATTERN/
```

这是匹配操作符，将`PATTERN`解释为一个正则表达式。操作符解析为一个双引号字符串而不是一个函数。参见第5章。

## map

```
map BLOCK LIST  
map EXPR, LIST
```

这个函数为`LIST`中的各个元素计算`BLOCK`或`EXPR`（将`$_`局部设置为各个元素），并返回由各个计算的结果组成的列表。在列表上下文中它会计算`BLOCK`或`EXPR`，所以`LIST`中的各个元素会映射到返回值中的0个、1个或多个元素。这些会扁平化为一个列表。例如：

```
@words = map { split " " } @lines;
```

将一个行列表分解为一个单词列表。不过在输入值和输出值之间通常存在一个一对一的映射：

```
@chars = map chr, @nums;
```

将一个数字列表转换为相应的字符。下面是一个一对二映射的例子：

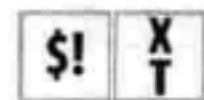
```
%hash = map { genkey($_) => $_ } @array;
```

这与下面的代码功能是一样的，不过前一种写法更有趣：

```
%hash = ();  
for my $_ (@array) {  
    $hash{genkey($_)} = $_;  
}
```

由于`$_`是列表值的一个别名（隐式引用），这个变量可以用来修改数组的元素。这是允许的，也很有用，不过如果`LIST`不是一个命名数组，这可能会导致奇怪的结果。要达到这个目的，使用一个常规的`foreach`循环可能更清晰。另外参见`grep`。`map`与`grep`的不同之处在于：`map`返回的列表由计算`EXPR`后的各个结果组成，而`grep`返回的列表由`LIST`中`EXPR`计算为`true`的各个值组成。

## mkdir



```
mkdir FILENAME, MASK  
mkdir FILENAME
```

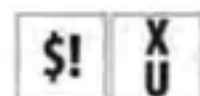


这个函数创建`FILENAME`指定的目录，为它授予`MASK`指定的权限，`MASK`数值可由当前`umask`修改。如果操作成功，返回`true`；否则返回`false`。

如果省略`MASK`，则假设掩码为`0777`，大多数情况下这就是你想要的掩码。一般地，与其提供一个受限的`MASK`而不允许用户有更多权限，不如先用权限`MASK`（如`0777`）创建目录，并允许用户用`umask`修改这个掩码。只有一种情况例外，文件或目录（例如，邮件文件）要保持私有时不要这样做。参见`umask`。

如果`mkdir(2)`系统调用不是你的C库的内置系统调用，Perl会通过为各个目录调用`mkdir(1)`程序来模拟。如果你要在这样一个系统上创建大量目录，更高效的做法是对这个目录列表自行调用`mkdir`程序，而不是启动过多的子进程。

## msgctl

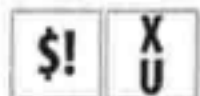


`msgctl ID, CMD, ARG`

这个函数调用System V IPC `msgctl(2)`系统调用；参见`msgctl(2)`了解更多详细信息。可能必须先声明`use IPC::SysV`才能得到正确的常量定义。如果`CMD`为`IPC_STAT`，则`ARG`必须是一个变量，将包含返回的`msqid_ds` C结构。返回值与`ioctl`和`fcntl`类似：`undef`表示错误，“0 but true”表示0，或者是具体的返回值。

这个函数只在支持System V IPC的机器上可用，这种机器比支持套接字的机器少得多。

## msgget

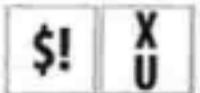


`msgget KEY, FLAGS`

这个函数调用System V IPC `msgget(2)`系统调用。另外参见`msgget(2)`了解更多详细信息。这个函数返回消息队列ID，或者如果有一个错误则返回`undef`。调用之前，应当声明`use IPC::SysV`。

这个函数只在支持System V IPC的机器上可用。

## msgrcv

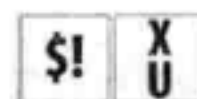


`msgrcv ID, VAR, SIZE, TYPE, FLAGS`

这个函数调用`msgrcv(2)`系统调用，从消息队列`ID`接收一个消息放在变量`VAR`中，这个变量的最大消息大小为`SIZE`。参见`msgrcv(2)`了解更多详细信息。接收到一个消息时，`VAR`中首先是消息类型，`VAR`的最大长度为`SIZE`加上消息类型的大小。可以用`unpack("l!a*")`对它解码。如果成功，这个函数返回`true`，否则如果存在一个错误则返回`false`。调用之前应当先声明`use IPC::SysV`。

这个函数只在支持System V IPC的机器上可用。

## msgsnd



`msgsnd ID, MSG, FLAGS`

这个函数调用`msgsnd(2)`系统调用，将消息`MSG`发送到消息队列`ID`。参见`msgsnd(2)`了解更多信息。`MSG`必须以长整型消息类型开头。可以如下创建一个消息：

```
$msg = pack "l!a*", $type, $text_of_message;
```

如果成功，这个函数返回`true`，否则如果存在一个错误则返回`false`。调用之前应当先声明`use IPC::SysV`。

这个函数只在支持System V IPC的机器上可用。

## my

```
my TYPE EXPR : ATTRIBUTES
my EXPR : ATTRIBUTES
my TYPE EXPR
my EXPR
```

这个操作符声明只在最内层外围块、子例程、`eval`或文件中存在的一个或多个私有变量。如果列出了多个变量，这个列表必须放在小括号中，因为这个操作符比逗号绑定更紧密。只有简单标量或完整的数组和散列可以采用这种方式声明。

这个变量名不能用包名限定，因为包变量都可以通过其相应的符号表全局访问，而词法作用域变量与任何符号表都无关。不同于`local`，这个操作符与全局变量没有关系，并不是隐藏同名的其他变量，使它们在其作用域（也就是这个私有变量所在的作用域）中不可见。与之不同，全局变量总可以通过其包限定形式访问，或者通过一个符号引用来访问。

私有变量的作用域要从这个变量声明后面的语句开始。变量的作用域延伸到在此之后的所有外围块，直到这个变量本身的作用域结束。

不过，这表示从私有变量作用域调用的任何子例程不会看到这个私有变量，除非定义子例程本身的块包围在这个变量的作用域中。听上去很复杂，不过一旦掌握这个概念，你就会发现实际上并不复杂。相应的技术术语是建立词法作用域（lexical scoping），所以我们通常称这些变量为词法作用域变量（lexical variables）。在C文化中，它们有时被称为“自动”变量，因为在进入和退出作用域时它们会自动分配和撤销。

如果需要，可以为`EXPR`赋值，这就允许初始化你的词法作用域变量（如果没有给定初始化代码，所有标量都初始化为未定义值，所有数组和散列初始化为空列表）。与正常的赋值一样，如果对左边的变量使用了小括号（或者如果变量是一个数组或散列），右边的表达



式将在列表上下文中计算。否则，右边的表达式会在标量上下文中计算。例如，可以利用一个列表赋值对子例程形参命名，如下所示：

```
my ($friends, $romans, $countrymen) = @_;
```

不过，要当心不要省略列表赋值的小括号，如下所示：

```
my $country = @_; # 对还是错？
```

这会把数组的长度（也就是子例程的参数个数）赋给这个变量，因为数组是在标量上下文中计算。不过，对于形参，只要使用`shift`操作符，可以充分利用标量赋值。实际上，由于对象方法会传入对象作为第一个参数，很多方法子例程开始时都会“盗取”第一个参数：

```
sub simple_as {  
    my $self = shift; # 标量赋值  
    my ($a,$b,$c) = @_; # 列表赋值  
    ...  
}
```

如果想用`my sub`声明一个词法作用域子例程，Perl会拒绝，并给出警告消息，指示这个特性尚未实现（当然，除非这个特性已经实现<sup>注8</sup>）。

`TYPE`和`ATTRIBUTES`是可选的。使用`TYPE`和`ATTRIBUTES`的声明可能如下所示：

```
my Dog $spot :ears(short) :tail(long);
```

如果指定了`TYPE`，这会指示`EXPR`中声明的标量是何种类型，可以是直接的一个或多个标量变量，或者间接地通过一个数组或散列指定。如果`TYPE`是类名，则认为这些标量包含该类型对象的引用，或者包含与这个类型兼容的对象的引用。特别要说明的是，派生类就认为是兼容的。也就是说，假设`Collie`派生自`Dog`，可以声明：

```
my Dog $lassie = new Collie;
```

这个声明指出你会把`$lassie`对象当作一个`Dog`对象来使用。尽管它实际上是一个`Collie`对象，但这一点并不重要，只要你只打算做`Dog`的工作。通过神奇的虚拟方法，那些`Dog`方法的实现也同样适用于`Collie`类，不过上面的声明只谈到了接口，而没有谈到实现。至少理论上是这样。

实际上，Perl并没有充分使用这个类型信息，不过这个信息可以用于将来的改进（以前伪散列曾经使用过这个信息，不过如今伪散列已经废弃不用了）。应当认为`TYPE`声明是一个通用类型接口，将来可能会根据不同的类以不同的方式实例化。事实上，`TYPE`甚至可能不是一个正式的类型名。我们为Perl保留了小写的类型名，因为扩展类型接口的一种方法就是

---

注8： 已经有希望了，因为Perl 6已经展示了子例程可以默认地设置为词法作用域，而且使用仍很容易。



允许可选的底层类型声明，如`int`、`num`和`str`<sup>注9</sup>。这些声明并不用来实现强类型机制，对于编译器来说，它们只是一些提示，指出大多数情况下都会按声明的方式使用这个变量，可以以此来优化这个变量的存储。标量的语义还是一样的，你仍然能让两个`str`标量相加，或者打印一个`int`标量，就好像它们是你熟悉的普通多态标量一样。不过，如果有一个`int`声明，Perl就会决定只存储整数值，而不再像现在这样缓存得到的字符串。使用`int`循环变量的循环可能运行得更快，特别是那些编译为C的代码。具体地，数字数组可以更紧凑地存储。作为一种极限情况，如果我们写如下的声明，内置`vec`函数甚至可能会过时：

```
my bit @bitstring;
```

`ATTRIBUTES`声明比类型使用得更多，参见第29章的`attributes pragma`了解更多有关内容。将来我们很可能还会实现一个属性`constant`：

```
my num $PI : constant = atan2(1,1) * 4;
```

不过，还有很多其他可能性，如为数组和散列建立默认值，或者允许变量在合作的解释器之间共享。类似于类型接口，属性接口也应当认为是一个通用接口，是一种发明新语法和语义的工作台。我们不知道Perl在未来十年会如何发展，我们只知道通过提前规划可以让这更容易一些。

参见`local`、`our`和`state`，以及第4章的“作用域声明”一节。

## new

```
new CLASSNAME LIST
new CLASSNAME
```

并没有内置的`new`函数。这只是一个普通的构造方法（也就是说，是一个用户自定义的子例程），由`CLASSNAME`类（即包）定义或继承，允许你构造`CLASSNAME`类型的对象。很多构造函数都名为“`new`”，不过这只是一个习惯（而不是规范），可能会让C++程序员错误地以为他们知道在做什么。一定要好好阅读当前类的文档，了解如何调用其构造函数；例如，在Tk部件集中创建列表框的构造函数就是`Listbox`。另外参见第12章。

## next

\$@

```
next LABEL
next
```

`next`操作符就像C中的`continue`语句：对于`LABEL`指定的循环，`next`操作符会启动这个循环的下次迭代：

注9：实际上，当前在Perl 6中只是为这些原生类型建立了这个语法的原型，一旦使用Perl 6的人发现不好的东西，使用Perl 5的人可能还会返回来用原先好的特性。

```

LINE: while (<STDIN>) {
    next LINE if /^#/; # 丢掉注释
    ...
}

```

如果这个例子中有一个`continue`块，它会在调用`next`之后立即执行。省略`LABEL`时，这个操作符则指示最内层外围循环。

块本身在语义上等同于只执行一次的循环。因此，`next`会提前退出这个块（通过`continue`块，如果有的话）。

不能用`next`退出一个有返回值的块，如`eval {}`、`sub {}`或`do {}`，另外不能用于退出一个`grep`或`map`操作。如果启用了警告，倘若用`next`退出一个不在当前词法作用域中的循环，如一个调用子例程中的循环，Perl就会发出警告。参见第4章“循环语句”一节。

## no

\$@

```

no MODULE VERSION LIST
no MODULE VERSION
no MODULE LIST
no MODULE
no VERSION

```

参见`use`操作符，从某种程度上讲，它与`no`正好相反。大多数标准模块不会取消导入任何东西，这使得`no`几乎成为一个无实际工作的操作（no-op）。实现Pragma的模块更有约束性。如果没有找到`MODULE`，会产生一个异常。

## oct

\$\_

```

oct EXPR
oct

```

这个函数将`EXPR`解释为一个八进制字符串，并返回等价的十进制值。如果`EXPR`以“`0x`”开头，将解释为一个十六进制字符串。如果`EXPR`以“`0b`”开头，将解释为一个二进制位串。下面的代码使用标准Perl记法转换为十进制、二进制、八进制和十六进制的整数输入串：

```
$val = oct $val if $val =~ /^0/;
```

要完成相反的功能，可以采用一种合适的格式使用`sprintf`：

```

$dec_perms = (stat("filename"))[2] & 07777;
$oct_perm_str = sprintf "%o", $perms;

```

`oct`函数常用于转换文件模式，如将类似“644”的数据串转换为一个文件模式。尽管Perl会根据需要自动将字符串转换为数字，但是这种自动转换采用的是十进制。

前导空白符会被忽略而没有任何警告，末尾如果有非数字字符，也会被忽略而且没有任何警告，如小数点（oct只处理非负整数，不能处理负整数或浮点数）。

## open



```
open FILEHANDLE, MODE, EXPR, LIST
open FILEHANDLE, MODE, EXPR
open FILEHANDLE, MODE, REFERENCE
open FILEHANDLE, EXPR
```

open函数将一个内部FILEHANDLE与EXPR或LIST给定的一个外部文件规范相关联。调用时可以提供两个或3个参数（或者如果第3个参数是一个命令，还可以有更多参数）。有3个或更多参数时，第2个参数指定了打开文件时的访问模式（MODE），其余的参数提供了具体的文件名或要执行的命令（这取决于模式）。如果指定了命令，倘若想直接调用这个命令而不调用shell，就像system或exec一样，还可以提供额外的参数。或者也可以把命令作为一个参数（第3个参数）提供给open，在这种情况下，是否调用shell取决于命令是否包含shell元字符（如果参数是普通的文件名，不要使用超过3个参数；否则会有问题）。如果无法识别MODE，open会产生一个异常。

作为一种特殊情况，如果3参数形式有一个读/写模式，另外第3个参数为undef：

```
open(my $tmp, "+>", undef) or die ...
```

这会打开一个匿名临时文件的文件句柄。对称地也可以使用“+<”，不过应该首先考虑向这个临时文件写点什么。需要使用seek定位来完成读操作。

可以使用3参数形式的open来指定I/O层（有时称为“准则”），应用到这个句柄，这会影响如何处理输入和输出（有关的更多详细信息参见PerlIO模块）。例如：

```
open(my $fh, "< :encoding(UTF-8)", "filename")
|| die "can't open UTF-8 encoded filename: $!";
```

这会打开一个UTF-8编码的文件（也就是一个包含Unicode字符的文件）。在v5.14中，UTF-8输入流上的默认行为是不会对编码错误抛出异常。如果使用UTF-8层，可以考虑增加：

```
use warnings FATAL => "utf8";
```

这样就能捕获到异常。另外参见第6章。

需要说明，如果在3参数形式中指定了层，则会忽略存储在\${^OPEN}中的默认层（参见第25章，默认层由open pragma或开关-CioD设置）。



如果你的Perl使用PerlIO<sup>注10</sup>构建，可以直接对Perl标量打开文件句柄，为此要传递该标量的一个引用作为3参数形式中的*EXPR*参数：

```
open($fh, ">", \ $variable) || ...
```

要重新打开STDOUT或STDERR作为一个内存中文件，首先要将它关闭：

```
close(STDOUT) || die "can't close STDOUT: $!";  
open(STDOUT, ">", \ $variable) || die "can't memopen STDOUT: $!";
```

如果只有两个参数，则认为模式和文件名/命令合并在第2个参数中（如果第二个参数中没有指定模式，只是一个文件名，那么出于安全的考虑，文件会以只读方式打开）。

```
open(LOG, "> logfile") or die "Can't create logfile: $!"; # 好  
open(LOG, ">", "logfile") or die "Can't create logfile: $!"; # 更好
```

`open`函数成功时会返回true，否则返回undef。如果`open`打开了到一个子进程的管道，返回值将是这个新进程的进程ID。与所有系统调用一样，一定要检查`open`的返回值，以确保它确实能正常工作<sup>注11</sup>。不过这不是C或Java，所以如果能用`||`操作符就不要使用if语句。还可以使用`or`，如果确实使用`or`，可以省略`open`上的小括号。省略函数调用上的小括号时，这会将它变成一个列表操作符，要当心在列表后面要使用“`or die`”而不是“`|| die`”。这是因为`||`的优先级要高于类似`open`的列表操作符，因此有可能得到出乎意料的结果：`||`会绑定到最后一个参数，而不是整个`open`：

```
open LOG, ">", "logfile" || die "Can't create logfile: $!"; # 不正确  
open LOG, ">", "logfile" or die "Can't create logfile: $!"; # 正确
```

这看起来不是太好，所以最好使用小括号，以便更清楚地看出列表操作符在哪里结束：

```
open(LOG, ">", "logfile") or die "Can't create logfile: $!"; # 好  
open(LOG, ">", "logfile") || die "Can't create logfile: $!"; # 好
```

或者只需把`or`放在另一行上：

```
open LOG, ">", "logfile"  
or die "Can't create logfile: $!";
```

如以上例子所示，*FILEHANDLE*参数通常只是一个简单的标识符（通常全大写），不过它也可以是一个表达式，这个表达式的值提供实际文件句柄的一个引用（这个引用可能是文件句柄名的一个符号引用，或者是任何可以解释为一个文件句柄的对象的硬引用）。这称为间接文件句柄（indirect filehandle），任何取*FILEHANDLE*作为第一个参数的函数除了可以处理直接文件句柄外，也可以处理这些间接文件句柄。不过，`open`很特殊：如果提供一个未定义的变量表示间接文件句柄，Perl会为你自动定义该变量。也就是说，自动生成这个

---

注10：2002年v5.8版本以来的默认构建配置。

注11：除非你使用了`autodie pragma`，它会负责为你完成检查。

变量，使它包含一个适当的文件句柄引用。这样做的一个好处是，如果一个文件句柄没有更多的引用指向它（通常是在变量超出作用域时），它就会自动关闭：

```
{
    my $fh;                                # (uninitialized)
    open $fh, ">", "logfile"               # $fh是自动生成的
        or die "Can't create logfile: $!";
    ...                                    # 对$fh做一些处理
}                                           # $fh在这里关闭
```

my \$fh声明可以结合到open中：

```
open(my $fh, ">", "logfile") || die ...
```

你已经见过文件名前面的>符号，这是模式的一个例子。模式可能作为文件名参数的一部分，也可能是文件名前面的一个参数。历史上，最先出现的是两参数形式的open。新增加的3参数形式允许将模式与文件名分开，这有一个好处，可以避免二者之间可能出现的混淆。在下面的例子中，我们知道用户不想打开一个正好以“>”开头的文件名。可以确信他指定MODE为“>”，这会打开EXPR中指定的文件完成写操作，如果这个文件不存在则会创建文件，如果文件确实已经存在，则截断这个文件的所有内容：

```
open(LOG, ">", "logfile") || die "Can't create logfile: $!";
```

采用短形式时，文件名和模式在同一个字符串中。这个字符串的解析就类似于典型的shell处理文件和管道重定向。首先，从字符串中去除所有前导和末尾的空白符。然后检查这个字符串（如果需要，两端都要检查），查看是否有指定文件打开模式的字符。模式和文件名之间允许有空白符。

指示如何打开文件的模式是类shell的重定向符号。表27-3提供了这样一组符号。如果要用这个表中没有提到的其他模式组合来访问文件，参见底层的sysopen函数。

表27-3：打开模式

	读模式	写访问	追加访问	创建不存在的文件	截断现有的文件
< PATH	Y	N	N	N	N
> PATH	N	Y	N	Y	Y
>> PATH	N	Y	Y	Y	N
+< PATH	Y	Y	N	N	N
+> PATH	Y	Y	N	Y	Y
+>> PATH	Y	Y	Y	Y	N
COMMAND	N	Y	n/a	n/a	n/a
COMMAND	Y	N	n/a	n/a	n/a



如果模式是“<”或什么都没有，会打开一个现有文件完成输入。如果模式为“>”，会打开文件完成输出，这会截断现有文件，或者创建不存在的文件。如果模式是“>>”，则会根据需要创建文件，并打开这个文件完成追加，所有输出会自动放在该文件的末尾。如果由于使用了一个模式“>”或“>>”而必须创建一个文件，新文件上的访问权限取决于进程的当前umask，将遵循该函数的有关规则。

下面是一些常见的例子：

```
open(INFO,      "datafile") || die("can't open datafile: $!");
open(INFO,      "< datafile") || die("can't open datafile: $!");
open(RESULTS,   "> runstats") || die("can't open runstats: $!");
open(LOG,        ">> logfile ") || die("can't open logfile: $!");
```

如果想要标点符号少一些的版本，可以写为：

```
open(INFO,      "datafile") or die "can't open datafile: $!";
open(INFO,      "< datafile") or die "can't open datafile: $!";
open(RESULTS,   "> runstats") or die "can't open runstats: $!";
open(LOG,        ">> logfile ") or die "can't open logfile: $!";
```

打开文件完成读操作时，特殊文件名“-”表示STDIN。打开文件用于写时，这个特殊文件名则表示STDOUT。正常情况下，它们分别指定为“<-”和“>-”。

```
open(INPUT, "-") || die; # 重新打开标准输入用于读
open(INPUT, "<-") || die; #作用相同，不过更明确
open(OUTPUT, ">-") || die; #重新打开标准输出用于写
```

这样一来，用户可以为程序提供一个文件名，它将使用标准输入或标准输出，不过程序的作者不用编写特殊代码来了解这一点。

还可以在这3个模式前面加一个“+”来请求同时读写。不过，不论是截断还是创建文件，另外不论它是否必须已经存在，这仍然由你选择的小于或大于号控制。这说明“+<”几乎总是倾向于完成读/写更新，不确定的“+>”模式则在你试图从文件中读取内容之前先截断文件（只有当你想重新读取刚刚写入的内容时，才会使用这种模式）。

```
open(DBASE, "+< database")
|| die "can't open existing database in update mode: $!";
```

可以把一个为了完成更新而打开的文件看作是一个随机访问数据库，可以使用seek移动某个特定的字节数，不过由于常规文本文件往往有变长的记录，这使得使用读写模式来更新这种文件是不可行的。参见第17章的-i命令行选项，其中介绍了另外一种完成更新的方法。

如果EXPR中的前导字符是一个管道符号，open会建立一个新进程，并将一个只写的文件句柄连接到该命令。这样一来，你可以写入这个句柄，而且写入的内容会出现在这个命令的标准输入中。



例如：

```
open(PRINTER, "| lpr -Plp1") || die "can't fork: $!";
say PRINTER "stuff";
close(PRINTER) || die "lpr/close failed: $?/$!";
```

如果`EXPR`的末尾字符是一个管道符号，`open`同样会启动一个新进程，不过这一次会连接一个只读的文件句柄。这样一来，命令写入其标准输出的所有内容都会出现在你的句柄上，可供读取。例如：

```
open(NET, "netstat -i -n |") || die "can't fork: $!";
while (<NET>) { ... }
close(NET) || die "can't close netstat: $!/$?";
```

如果显式关闭所有管道文件句柄，会导致父进程等待子进程完成，并在`$?` (`$CHILD_ERROR`)中返回状态码。`close`也可以设置`!($OS_ERROR)`。参见`close`和`system`下的例子，了解如何解释这些错误码。

包含shell元字符（如通配符或I/O重定向）的所有管道命令会传递到你的系统的标准shell（在Unix上是`/bin/sh`），所以这些shell特定的构造可以先处理。如果没有找到元字符，Perl会自行启动新进程而不调用shell。

你还可以使用3参数形式来启动管道。如果使用这种风格，之前的管道式`open`可以等价地写为：

```
open(PRINTER, "|-", "lpr -Plp1") || die "can't fork: $!";
open(NET, "-|", "netstat -i -n") || die "can't fork: $!";
```

在这里，第2个参数中的负号表示第3个参数中的命令。这些命令不会调用shell，但是如果你想确保完全没有shell处理出现，新版本的Perl允许你写为：

```
open(PRINTER, "|-", "lpr", "-Plp1") || die "can't fork: $!";
open(NET, "-|", "netstat", "-i", "-n") || die "can't fork: $!";
```

如果使用两参数形式来打开一个管道与特殊命令“`-`”<sup>注12</sup>交互，会首先完成一个隐式的`fork`（在不能执行`fork`的系统上，这会产生一个异常。20世纪的Microsoft系统大多不支持`fork`，不过在那之后已经可以支持了）。在这里，负号表示你的新子进程，这是父进程的一个副本。对于这个派生子进程的`open`，其返回值取决于谁在查看；从父进程检查时会返回子进程的进程ID；如果从子进程检查，则是0；如果`fork`失败，则返回未定义的值`undef`，在这种情况下，没有子进程。例如：

```
my $pid = open(FROM_CHILD, "-|") // die "can't fork: $!";
```

---

注12： 或者可以把它看作是之前3参数形式的一种特殊形式，这里省略了命令参数。

```

if ($pid) {
    @parent_lines = <FROM_CHILD>; # 父进程代码
}
else {
    print STDOUT @child_lines;    # 子进程代码
    exit;
}

```

对于父进程，文件句柄的表现是正常的，不过对于子进程，父进程的输入（或输出）会通过管道从子进程的STDOUT传出或者传入子进程的STDIN。子进程不会看到父进程的文件句柄已经打开（可以由PID为0来指示）。

一般地，如果希望能更多地控制管道命令如何执行（如运行setuid时），而且不想扫描shell命令中是否有元字符，此时就应当使用这个构造而不是正常的管道式open。下面的管道式open：

```

open(FH, "| tr 'a-z' 'A-Z'");           # 管道到shell命令
open(FH, "|-", "tr", "a-z", "A-Z" );    # 管道到裸命令
open(FH, "|-") || exec("tr", "a-z", "A-Z") || die; # 管道到子进程
open(F00, "|-", "tr", "a-z", "A-Z")      || die; # 管道到子进程

```

与以下构造基本上等价：

```

open(FH, "cat -n 'file' |");             # 来自shell命令的管道
open(FH, "-|", "cat", "-n", "file");     # 来自裸命令的管道
open(FH, "-|") || exec("cat", "-n", "file") || die; # 来自子进程的管道
open(F00, "-|", "cat", "-n", $file)      || die; # 来自子进程的管道

```

各个块中的最后两个例子显示了管道的“列表形式”，并不是所有平台上都支持这一点。对此有一个很好的经验：如果你的平台支持真正的fork（换句话说，如果你的平台是Unix），就可以使用这种列表形式。

有关的更多例子见第15章“匿名管道”一节。要了解fork open更复杂的用法，参见第15章“自言自语”一节和第20章“清理环境”一节。

在任何可能完成一个fork的操作之前，Perl会刷新输出（flush）所有打开来完成输出的文件，不过在某些平台上可能不支持这一点。为了保证安全，可能需要设置\$I（使用English模块则是\$AUTOFLUSH），或者在打开的句柄上调用IO::Handle的autoflush方法。

在支持文件“执行时关闭”（close-on-exec）标志的系统上，会为新打开的文件描述符设置这个标志，由\$^F（\$SYSTEM\_FD\_MAX）的值确定。

关闭管道文件句柄会导致父进程等待子进程完成，并在\$?和\${^CHILD\_ERROR\_NATIVE}中返回状态值。

对于两参数形式的`open`，传入文件名时，会删除所有前导和末尾空白符，而且接受常规的重定向符号。这个属性称为“魔法打开”，通常可以用来得到很好的效果。用户可以指定一个文件名为“`rsh cat file |`”，或者可以根据需要改变某些文件名：

```
$filename =~ s/(.*\.\gz)\s*$/gzip -dc < $1|/;
open(FH, $filename) || die "Can't open $filename: $!";
```

用`open`启动一个命令时，必须选择输入还是输出：“`cmd|`”用于读，或者是“`|cmd`”用于写。可能不会使用`open`来启动一个同时通过管道输入输出的命令，例如“`|cmd|`”（目前）就是一种不合法的记法。不过，标准`IPC::Open2`和`IPC::Open3`库例程提供了一个很接近的等价构造。关于双端管道的详细信息，参见第15章的“双向通信”一节。

按照Bourne shell传统，你还可以指定一个以`>&`开头的`EXPR`，在这种情况下，字符串的其余部分解释为一个文件句柄的名字（或者如果是数值，则解释为文件描述符），这是要利用`dup2(2)`系统调用<sup>注13</sup>复制的文件句柄。可以在`>`，`>>`，`<`，`+`，`++`和`+<`后面使用`&`（指定的模式应当与原文件句柄的模式匹配）。

之所以希望这样做，可能是因为你已经有一个打开的文件句柄，并且希望建立另一个文件句柄，而它实际上是第一个句柄的一个副本。

```
open(SAVEOUT, ">&SAVEERR") || die "couldn't dup SAVEERR: $!";
open(MHCONTEXT, "<&4") || die "couldn't dup fd4: $!";
```

这意味着如果一个函数需要一个文件名参数，但你已经有一个打开的文件因而不想它提供一个文件名，就可以直接传递这个文件句柄加一个前导`&`字符。不过，最好使用一个完全限定句柄，以免函数恰好在另外一个不同的包中：

```
somefunction("&main::LOGFILE");
```

使用“`dup`”复制文件句柄的另一个原因是可能要临时重定向一个现有的文件句柄，而不丢失原来的目标。下面的脚本会保存、重定向并恢复`STDOUT`和`STDERR`：

```
#!/usr/bin/perl
open SAVEOUT, ">&STDOUT";
open SAVEERR, ">&STDERR";

open(STDOUT, "> foo.out") || die "Can't redirect stdout";
open(STDERR, ">&STDOUT") || die "Can't dup stdout";

select STDERR; $| = 1;      # 启用自动刷新
select STDOUT; $| = 1;      # 启用自动刷新

say STDOUT "stdout 1";      # 这些I/O流也传播到
say STDERR "stderr 1";      # 子进程
```

---

注13：（目前）这不适用于由文件句柄自动生成创建的匿名句柄，不过完全可以使用`fileno`来获取文件描述符并复制该描述符。



```

system("some command");      # 使用新的stdout/stderr

close STDOUT;
close STDERR;

open STDOUT, ">&SAVEOUT";
open STDERR, ">&SAVEERR";

say STDOUT "stdout 2";
say STDERR "stderr 2";

```

如果文件句柄或描述符号前面是`&=`组合而不是一个简单的`&`，那么并不是创建一个全新的文件描述符，Perl会利用`fdopen(3)` C库调用使这个`FILEHANDLE`作为现有描述符的一个别名。这样更节省系统资源，不过如今这已经不像从前那么重要了。

```

$fd = $ENV{"MHCONTEXTFD"};
open(MHCONTEXT, "<&=$fdnum")
    || die "couldn't fdopen descriptor $fdnum: $!";

```

文件句柄`STDIN`、`STDOUT`和`STDERR`会跨`exec`保持打开状态。默认情况下其他文件句柄不是这样。在支持`fcntl`函数的系统上，可以修改一个文件句柄的“执行时关闭”（`close-on-exec`）标志。

```

use Fcntl qw(F_GETFD F_SETFD);
$flags = fcntl(FH, F_SETFD, 0)
    || die "Can't clear close-on-exec flag on FH: $!";

```

参见第25章中的特殊变量`^F`（`$SYSTEM_FD_MAX`）。

如果利用两参数形式的`open`，使用一个字符串变量作为文件名时必须当心，因为这个变量可能包含任意的奇怪字符（特别是文件名可能由互联网上奇怪的人提供）。如果不小心，文件名的某些部分就可能解释为一个模式（`MODE`）串、不可忽略的空白符、重复的规范或者负号。以前可能会利用下面这种有趣的方法来保证安全：

```

$path =~ s#^\s#.#/$1#;
open(FH, "< $path\0") || die "can't open $path: $!";

```

不过这在很多方面还是会出问题。使用3参数形式的`open`可以妥善地打开任意的文件名，而不存在（额外的）安全风险：

```

open(FH, "<", $path) || die "can't open $path: $!";

```

另一方面，如果你要找一个真正的C风格的`open(2)`系统调用，包括它的所有相关问题，可以参考`sysopen`：

```

use Fcntl;
sysopen(FH, $path, O_RDONLY) || die "can't open $path: $!";

```

如果你的系统区分文本和二进制文件，可能需要将文件句柄置于二进制模式（或者干脆放弃，这要看具体情况而定），以避免毁坏你的文件。在这些系统上，如果你在一个二进制

文件上使用文本模式，或者在一个文本文件上使用二进制模式，你可能不会喜欢所得到的结果。

需要binmode函数的系统与不需要binmode函数的系统在文本文件所用的格式方面有很大不同。不需要这个函数的系统用一个字符结束每一行，这个字符对应于C所认为的一个换行符\n。Unix（包括现代版本的Mac OS）都属于这一类。VMS、MVS、MSwhatever，以及其他类型的S&M操作系统对于文本文件和二进制文件的I/O有不同的处理，所以它们需要binmode。

或者可以使用它的等价构造。可以在open函数中指定二进制模式，而不需要一个单独的binmode调用。作为MODE参数的一部分（不过只适用于3参数形式），可以指定不同的输入和输出层。要完成等价的binmode，可以使用3参数形式的open，在其他MODE字符后面提供一个:raw层：

```
open(FH, "< :raw", $path) || die "can't open $path: $!";
```

至于这里还可以指定什么输入和输出层，有关的更多详细内容参见第6章的Encode模块，包括Windows文本文件的处理。

## opendir

\$!	X	X	X
ARG	ARG	T	U

```
opendir DIRHANDLE, EXPR
```

这个函数打开一个名为EXPR的目录，可由readdir、telldir、seekdir、rewinddir和closedir处理。如果成功，这个函数返回true。目录句柄有自己的命名空间，与文件句柄的命名空间不同。

参见readdir中的例子。

## ord

\$_
-----

```
ord EXPR  
ord
```

这个函数返回EXPR首字符的数字值（码点）。返回值总是无符号的。如果你希望得到有符号值，可以使用unpack("c", EXPR)。如果希望字符串中的字符转换为一个数字列表，可以使用unpack("U\*", EXPR)。要得到一个字符的码点，可以使用charnames pragma的charnames::vianame函数，并提供字符名作为一个字符串。

## our

```
our TYPE EXPR : ATTRIBUTES  
our EXPR : ATTRIBUTES
```

```
our TYPE EXPR
our EXPR
```

`our`声明一个或多个变量在外围块、文件或`eval`中是合法的全局变量。也就是说，`our`与`my`声明在确定可见性方面遵循同样的规则，不过它不会创建一个新的私有变量，而是允许自由地访问现有的包全局变量。如果列出了多个值，这个列表必须放在小括号中。

`our`声明的主要用途是隐藏变量，使之不受`use strict "vars"`声明的影响，由于变量伪装成一个`my`变量，这就允许你使用声明的全局变量，而不需要用包名限定这个变量。不过，就像`my`变量一样，这只适用于`our`声明的词法作用域内。在这个方面，它与`use vars`有所不同，后者会影响整个包，而不只是词法作用域。

`our`在另一个方面也类似于`my`，同样可以结合`TYPE`和`ATTRIBUTES`来声明变量。语法如下：

```
our Dog $spot :ears(short) :tail(long);
```

写这本书时，它的含义还不是很清楚。属性可以影响`$spot`的全局或局部解释。一方面，它很像是`my`变量，属性可以调整`$spot`的当前局部视图，而不影响全局变量在其他地方的其他视图。另一方面，如果一个模块声明`$spot`是一个`Dog`，而另外一个模块声明`$spot`是一个`Cat`，你最后有可能得到一个喵喵叫的狗或汪汪叫的猫。这还是一个正在研究的课题，实际上这么讲只是说明我们还没有完全掌握这个方面。

`our`还有一点类似于`my`，就是它的可见性。`our`声明会声明一个全局变量，它在整个词法作用域中都是可见的，即使跨包边界也不影响。变量所在的包会在声明时确定，而不是在使用时确定。这说明，下面的行为是成立的，一般认为这是一个特性：

```
package Foo;
our $bar;           # $bar在词法作用域的其余部分就是$Foo::bar
$bar = 582;

package Bar;
print $bar;         # 打印582，就像"our"是"my"一样
```

不过，`my`会创建一个新的私有变量，而`our`提供一个已有的全局变量，这个区别很重要，特别是在赋值时。如果结合一个运行时赋值和`our`声明，一旦`our`出了作用域，这个全局变量的值不会消失。要让它消失，需要使用`local`：

```
($x, $y) = ("one", "two");
say "before block, x is $x, y is $y";
{
    our $x = 10;
    local our $y = 20;
    say "in block, x is $x, y is $y";
}
say "past block, x is $x, y is $y";
```

这会打印以下结果：



```
before block, x is one, y is two
in block, x is 10, y is 20
past block, x is 10, y is two
```

同一个词法作用域中允许有多个`our`声明，只要它们在不同的包中。如果它们恰好在同一个包中，Perl会对此发出警告（如果启用了警告）。

```
use warnings;
package Foo;
our $bar;                # 对词法作用域的其余部分声明$Foo::bar
$bar = 20;

package Bar;
our $bar = 30;           #对词法作用域的其余部分声明$Bar::bar
print $bar;              # 打印30

our $bar;                # 发出警告
```

参见`local`、`our`和`state`，以及第4章的“作用域声明”一节。

## pack

\$@

```
pack TEMPLATE, LIST
```

这个函数取正常Perl值的一个`LIST`，根据`TEMPLATE`将它们转换为一个字节串，并返回这个字节串。`pack`和`unpack`的模板在第26章介绍。

## package

```
package NAMESPACE VERSION BLOCK
package NAMESPACE VERSION
package NAMESPACE BLOCK
package NAMESPACE
```

实际上这并不是一个函数，而是一个声明，指出`BLOCK`或最内层外围作用域的其余部分属于指定的符号表或命名空间（因此`package`声明的作用域等同于`my`、`state`或`our`声明的作用域）。在这个作用域中，基于这些声明，编译器会在声明包的符号表中查找，来解析所有未限定的全局标识符。

`package`声明只影响全局变量，包括已经使用了`local`的那些全局变量，但不包括用`my`、`state`或`our`创建的词法作用域变量。它只影响未限定的全局变量，用其自己的包名限定的全局变量会忽略当前声明的包。用`our`声明的全局变量是非限定的，因此会采用当前包，不过仅限于声明时，在此之后它们表现得就像`my`变量一样。也就是说，对于其词法作用域的其余部分，`our`变量会在声明时“钉入”当前所用的包，即使后面又有`package`声明插进来也不影响。

通常，会把`package`声明作为第一条语句放在一个文件的最前面（将由`require`或`use`操作

符包含这个文件），不过只要能放语句的地方都可以放置package声明。创建一个传统的或面向对象的模块文件时，习惯做法是将包命名为与文件同名，以避免混淆（还有一个习惯，命名这些包时会以一个大写字母开头，因为按约定小写的模块会解释为实现Pragma的模块）。

可以从多个不同地方切换到一个给定的包，这只会影响编译器在块中其余的部分使用哪个符号表（如果编译器看到同一级上的另一个package声明，这个新的声明会覆盖原来的声明）。Perl假设你的主程序以一个不可见的package main声明开头。

如果提供了VERSION，package会把给定命名空间中的\$VERSION变量设置为有指定VERSION的version对象。VERSION必须是一个“strict”风格（由version pragma定义）的版本号：这是一个正小数（有整数或小数部分）但没有指数，或者是一个“带点数版本串”（dotted-decimal vstring），有一个前导v字符，另外至少有3个分量（随着人们越来越习惯于版本对象，这个要求将来可能会放宽为两个分量）。每个包应当只设置一次\$VERSION。

可以用包名和一个双冒号限定标识符，来指示另一个包中的变量、子例程、句柄和格式：`$Package::Variable`。如果包名为null，则假设为main包。也就是说，`$::sail`等价于`$main::sail`，也等价于`$main'sail`，有时会在较早的代码中看到这种写法。

下面给出一个例子：

```
package main;          $sail = "hale and hearty";
package Mizzen;        $sail = "tattered";
package Whatever;
say "My main sail is $main::sail.";
say "My mizzen sail is $Mizzen::sail.";
```

这会打印以下结果：

```
My main sail is hale and hearty.
My mizzen sail is tattered.
```

一个包的符号表存储在一个散列中，这个散列的名字以一个双冒号结尾。例如，main包的符号表名为`%main::`。所以现有的包符号`*main::sail`也可以通过`$main::{"sail"}`来访问。关于包的更多信息参见第10章。另外参见这一章前面的my来了解有关作用域的其他问题。

## `__PACKAGE__`

这是一个特殊的token，返回它所在的包的名字。参见第10章。

## pipe

\$!	X	X
ARG		U

pipe *READHANDLE*, *WRITEHANDLE*

类似于相应的系统调用，这个函数会打开一对连接的管道，参见`pipe(2)`。这个调用通常用在`fork`前面，在此之后，管道的读者应当关闭`WRITEHANDLE`，写者要关闭`READHANDLE`（否则，写者关闭时，管道不会向读者指示EOF）。如果建立了一个管道进程环，除非特别小心，否则很可能会出现死锁。另外需要说明，Perl的管道使用标准I/O缓冲，所以在每个输出操作之后，可能需要在你的`WRITEHANDLE`上设置`$|`（`$OUTPUT_AUTOFLUSH`），这取决于应用，参见`select`（输出文件句柄）。

类似于`open`，如果某个文件句柄未定义，这个文件句柄将会自动生成。

下面是一个小例子：

```
pipe(README, WRITEME);
unless ($pid = fork) { # 子进程
    defined($pid) || die "can't fork: $!";
    close(README);
    for $i (1..5) { print WRITEME "line $i\n" }
    exit;
}
$SIG{CHLD} = sub { waitpid($pid, 0) };
close(WRITEME);
@strings = <README>;
close(README);
print "Got:\n", @strings;
```

注意写者如何关闭读取端，另外读者如何关闭写入端。不能使用一个管道完成双向通信。要想建立双向通信，要么使用两个不同的管道，要么使用`socketpair`系统调用来实现。参见第15章中“管道”一节。

## pop

X
ARG

pop *ARRAY*  
pop

这个函数将一个数组当作堆栈来处理，弹出（删除）并返回数组的最后一个值，使数组缩短一个元素。如果省略`ARRAY`，这个函数在子例程和格式的词法作用域中会弹出`@_`；在文件作用域（通常是主程序）或者在`eval STRING`、`BEGIN {}`、`CHECK {}`、`UNITCHECKINIT {}`和`END {}`构造建立的词法作用域中会弹出`@ARGV`。它与以下代码有同样的效果：

```
$tmp = $ARRAY[$#ARRAY--];
```

或：

```
$tmp = splice @ARRAY, -1;
```



如果数组中没有元素，`pop`会返回`undef`（不过不要依赖于这一点来确定数组是否为空，因为数组可能包含`undef`值）。另外参见`push`和`shift`。如果想弹出多个元素，可以使用`splice`。

`pop`要求它的第一个参数为数组，而不是一个列表。如果想得到一个列表的最后一个元素，可以使用：

```
(LIST)[-1]
```

从v5.14开始，`pop`可以取一个未祝福的数组的引用，它会自动解引用。`pop`的这个方面还是试验性的。具体的行为在Perl将来的版本中有可能改变。

## pos

\$\_

```
pos SCALAR  
pos
```

这个函数返回`SCALAR`中的一个位置，即`SCALAR`上最后一个`m//g`搜索停止的位置。

它返回最后一个匹配之后的字符（码点）的偏移量（也就是说，它等价于`length($`)+length($&)`）。这是该字符串上下一个`m//g`搜索开始的偏移量。要记住，字符串起始位置的偏移量为0。需要说明，0是一个合法的匹配偏移量。`undef`指示搜索位置要重置（通常是由于匹配失败，不过也可能是因为在这个标量上还没有运行过匹配）。

例如，如果有：

```
$graffito = "fee fie foe foo";  
while ($graffito =~ m/e/g) {  
    say pos $graffito;  
}
```

这会打印2，3，7和11，这是“e”后各个码点的偏移量。可以为`pos`函数赋一个值，来指定下一个`m//g`从哪里开始：

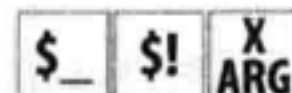
```
$graffito = "fee fie foe foo";  
pos $graffito = 4; # 跳过fee，从fie开始  
while ($graffito =~ m/e/g) {  
    say pos $graffito;  
}
```

这只会打印7和11。对于所搜索的字符串，正则表达式断言`\G`只匹配`pos`当前指定的位置。参见第5章中“位置”一节。

需要说明，我们所说的是码点，而不是字符。希望你不要将二者混淆。码点是程序员可见的字符，其中一些对用户来说可能是不可见的。用户可见的字符通常称为字形或字形簇，

可能由多个码点组成。例如，“`\r\n`”就是一个用户字符，但这是两个程序员字符。如果需要处理字形而不是码点的pos，可以参见CPAN `Unicode::GCString`模块。

## print



```
print FILEHANDLE LIST
print FILEHANDLE
print LIST
print
```

这个函数会打印一个字符串或用逗号分隔的字符串列表。如果设置了`$\`（`$OUTPUT_RECORD_SEPARATOR`），这个变量的内容会隐式打印在列表末尾。如果成功，这个函数返回`true`，否则返回`false`。各个`LIST`项之间会打印`$,`的当前值（如果有）。整个`LIST`打印完之后会打印`$\`的当前值（如果有）。

要单独使用`FILEHANDLE`向其中打印`$_`的内容，必须使用一个真正的文件句柄如`FH`，而不能是间接文件句柄如`$fh`。`FILEHANDLE`可以是一个标量变量名（无下标），在这种情况下，变量包含实际文件句柄的名字，或者包含某个文件句柄对象的引用。与所有其他间接对象一样，`FILEHANDLE`也可以是返回这样一个值的块：

```
print { $OK ? "STDOUT" : "STDERR" } "stuff\n";
print { $iohandle[$i] } "stuff\n";
```

如果`FILEHANDLE`是一个变量，而下一个token是一个项，可能会错误地把它解释为一个操作符，除非加入一个`+`，或者在参数两边加上小括号。

例如：

```
print $a - 2;    # 向默认文件句柄（通常是STDOUT）打印$a - 2
print $a (- 2); # 向$a中指定的文件句柄打印-2
print $a -2;    # 也打印-2（古怪的解析规则）
```

如果省略`FILEHANDLE`，这个函数会打印到当前选择的输出文件句柄，初始为`STDOUT`。要把默认文件句柄设置为`STDOUT`以外的另外某个文件句柄，可以使用`select FILEHANDLE`操作<sup>注14</sup>。如果`LIST`也被省略，函数会打印`$_`。因为`print`取一个`LIST`，所以`LIST`中的所有元素都会在列表上下文中计算。因此，如果有：

```
print OUT <STDIN>;
```

它不是从标准输入打印下一行，而是会打印标准输入的所有其余的行，直到文件末尾，因为这些都是`<STDIN>`在列表上下文中返回的内容。如果不希望这样，可以写为：

```
print OUT scalar <STDIN>;
```

---

注14：因此，`STDOUT`实际上并不是`print`的默认文件句柄。它只是默认的默认文件句柄。

另外，要记住这样一个规则：“如果看上去像是一个函数，那么它就是一个函数。”当心不要在`print`关键字后面加左括号，除非你希望用对应的右括号结束`print`的参数。应当加入一个`+`，或者在所有参数两边加小括号：

```
print (1+2)*3, "\n"; #不正确
print +(1+2)*3, "\n"; # 正确
print ((1+2)*3, "\n"); # 正确
```

如果指定一个`FILEHANDLE`，只有当`FILEHANDLE`是一个常规的裸字文件句柄而不是一个块或间接文件句柄时，才可以省略`LIST`。

```
$_ = "stuff\n";
*NEWOUT = *STDOUT;
print NEWOUT; #正确：打印"stuff\n"

$fh = *NEWOUT;
print $fh; #不正确：打印STDOUT "*main::STDOUT"
```

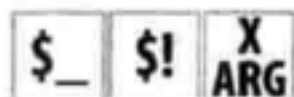
如果一个文件句柄没有一个I/O层指定如何编码，向这样一个文件句柄打印Unicode数据会触发一个强制性警告：“打印宽字符”。要修正这个问题，可以通过`binmode`指定编码，或者提供编码，作为3参数形式`open`的第二个参数。

```
binmode(STDOUT, ":utf8") || die "Can't binmode: $!";

open(HANDLE, "> :encoding(UTF-16)", $file)
|| die "Can't open $file: $!";
```

如果打印到一个关闭的管道或套接字，会生成一个SIGPIPE信号。参见第15章“信号”一节。

## printf



```
printf FILEHANDLE FORMAT, LIST
printf FORMAT, LIST
printf FILEHANDLE
printf LIST
printf
```

这个函数向`FILEHANDLE`打印一个格式化字符串，如果省略`FILEHANDLE`，则向当前选择的输出文件句柄（初始为`STDOUT`）打印格式化字符串。`LIST`中的第一项必须是一个字符串，指出如何格式化其余的项。这类似于C库的`printf(3)`和`fprintf(3)`函数。这个函数等价于：

```
print FILEHANDLE sprintf FORMAT, LIST
```

只不过不会追加`$\ ($OUTPUT_RECORD_SEPARATOR)`。

参见第26章了解如何解释格式。我们很想在这里再做一个详细的说明，不过这本书实在太庞大了，实在不允许我们这样做。



如果将一个非法的引用类型用作为`FILEHANDLE`参数，会产生一个异常。

如果`FORMAT`和`LIST`都省略，会使用`$_`。不过，在这种情况下，更应该使用`print`。不要落入这样一个陷阱：本来利用一个简单的`print`就可以完成的事情却使用了一个`printf`。`print`函数更高效，而且更不容易出错。

## prototype



`prototype FUNCTION`

这个函数将函数的原型作为一个字符串返回（或者如果函数没有原型，则返回`undef`）。`FUNCTION`是要获取原型的某个函数的一个引用，或者是这个函数的函数名。

如果`FUNCTION`是一个以`CORE::`开头的字符串，其余部分则认为是一个Perl内置函数的名字，如果没有这样一个内置函数，会产生一个异常。如果这个内置函数不是可覆盖的（如`qw//`），或者其参数不能用一个原型表述（如`system`），这个函数会返回`undef`，因为这个内置函数实际上表现得并不像一个Perl函数。否则，会返回描述等价原型的字符串。

## push



`push ARRAY, LIST`

这个函数把`ARRAY`当作一个堆栈，并把`LIST`的值压入`ARRAY`的末尾。`ARRAY`长度会增加，即会加上`LIST`的长度。这个函数将返回这个新长度。`push`函数与下面的代码有同样的效果：

```
for my $value (listfunc()) {  
    $array[++$#array] = $value;  
}
```

或者：

```
splice @array, @array, 0, listfunc();
```

不过`push`更高效（对你和你的计算机来说都是如此）。可以使用`push`并结合`shift`来建立一个相当高效的移位寄存器或队列：

```
for (;;) {  
    push @array, shift @array;  
    ...  
}
```

参见`pop`和`unshift`。

从v5.14开始，`push`可以取一个未祝福的数组的引用，它会自动解引用。`Push`的这个方面还是试验性的。具体的行为在Perl将来的版本中有可能改变。

## q/STRING/

```
q/STRING/  
qq/STRING/  
qr/STRING/  
qw/STRING/  
qx/STRING/
```

通用引号。另外参见第2章“选择你自己的引号”一节。有关qx//上的状态标注，参见readpipe。关于qr//上的状态标注，参见m//。另外参见第5章中的“保持控制”一节。

## quotemeta

\$\_

```
quotemeta EXPR  
quotemeta
```

这个函数返回*EXPR*的值，其中所有非字母数据字符均加上反斜线（也就是说，在返回字符串中，所有不匹配/[A-Za-z\_0-9]/的字符前面都会加反斜线，而不论本地化环境设置是什么）。这是实现内插上下文（包括双引号字符串、反斜线和模式）中\Q转义的内部函数。

## rand

```
rand EXPR  
rand
```

这个函数返回一个大于或等于0而且小于*EXPR*值的伪随机浮点数（*EXPR*应当是正数）。如果省略*EXPR*，这个函数会返回介于0到1之间的一个浮点数（包括0，但不包括1）。rand会自动调用srand，除非srand已经调用过。参见srand。

要得到一个整数值，如掷骰子，可以结合使用这个函数和int，如下所示：

```
$roll = int(rand 6) + 1;      # $roll现在是1到6之间的一个数
```

由于Perl使用你自己的C库的伪随机数函数，如random(3)或drand48(3)，不能保证随机数分布的质量。如果需要更强的随机性，如用于加密，可能需要参阅有关random(4)的文档（如果你的系统有一个/dev/random或/dev/urandom设备）；CPAN模块Math::Random::Secure、Math::Random::MT::Perl和Math::TrulyRandom；或者参考关于伪随机数计算生成的一些好书，如Knuth的第2卷<sup>注15</sup>。

## read

\$!	T	X	X
ARG	RO		

```
read FILEHANDLE, SCALAR, LENGTH, OFFSET
```

---

注15：Knuth, D.E. 《The Art of Computer Programming, Seminumerical Algorithms》第2卷，第3版（Addison-Wesley公司1997年出版）。这本书必读。

```
read FILEHANDLE, SCALAR, LENGTH
```

这个函数尝试从指定的`FILEHANDLE`读取`LENGTH`个字符（表示码点，而不是字形）的数据，并读入变量`SCALAR`。这个函数返回读取的字符个数，或者如果到达文件末尾，则返回0。出现错误时返回`undef`。`SCALAR`会扩展或收缩为实际读取的长度。如果指定了`OFFSET`，这会确定在变量的哪个位置开始放置字符，因此可以向一个字符串中间读入数据。

要从文件句柄`FROM`将数据复制到文件句柄`TO`，可以写为：

```
while (read(FROM, $buf, 16384)) {  
    print TO $buf;  
}
```

注意这里所说的“字符”：取决于文件句柄的状态，可能会读取（8位）字节或字符。字节只是Perl中对值比较小的未解码码点的一种叫法。默认地，所有文件句柄都会处理字节。不过，举例来说，如果打开文件句柄时提供了`:utf8` I/O层，I/O就会处理UTF-8编码的Unicode字符，而不是字节。两参数形式的`binmode`、`open`中间的参数（即模式参数）或者`open pragma`也是类似的：对于这些情况，可以读取几乎任意字符。

与`read`相对的是`print`，它已经知道你想写的字符串的长度，而且可以写任意长度的字符串。不要错误地使用`write`，`write`只能结合`format`使用。

Perl的`read`函数使用标准I/O的`fread(3)`函数实现，所以实际的`read(2)`系统调用可以读取超过`LENGTH`个字节来填充输入缓冲区，而`fread(3)`可能会做多个`read(2)`系统调用来填充缓冲区。为了有更多的控制，可以使用`sysread`指定实际的系统调用。`read`和`sysread`调用不能混杂在一起使用，除非你想变魔术（或者想自寻烦恼）。

## readdir

\$!	T	X	X
		ARG	U

```
readdir DIRHANDLE
```

这个函数从`opendir`打开的一个目录句柄读取目录项（就是简单的文件名）。在标量上下文中，这个函数返回下一个目录项（如果有）；否则，会返回`undef`。在列表上下文中，它返回这个目录中其余的所有目录项，如果没有目录项，则返回一个`null`列表。例如：

```
opendir(THISDIR, ".") || die "serious dainbramage: $!";  
@allfiles = readdir THISDIR;  
closedir THISDIR;  
say "@allfiles";
```

这会把当前目录中的所有文件打印在一行上。如果你想避开“.”和“..”目录项，可以采用以下某种方法（选择你认为最可读的一种）：

```
@allfiles = grep { $_ ne "." && $_ ne ".." } readdir THISDIR;  
@allfiles = grep { !/^([.])?\.z/ } readdir THISDIR;
```



```
@allfiles = grep { !/^\.{1,2}\z/ } readdir THISDIR;
@allfiles = grep { !/^\. \. ?\z/, readdir THISDIR;
```

要避开所有.\*文件（如ls程序），可以写为：

```
@allfiles = grep { !/^\. /, readdir THISDIR;
```

如果只想得到文本文件，可以写为：

```
@textfiles = grep -T, readdir THISDIR;
```

不过要当心最后一项，因为如果不是当前目录，必须把目录部分“粘”回到readdir的结果中，如下：

```
opendir(THATDIR, $path) || die "can't opendir $path: $!";
@dotfiles = grep { /^\. / && -f } map { "$path/$_" } readdir(THATDIR);
closedir THATDIR;
```

在v5.12中，可以在while循环中使用一个裸readdir，这会在每次迭代时设置\$\_.还可以使用一个未定义的标量变量，这会利用一个匿名目录句柄自动生成这个变量。

```
my $dh; # 确保它是新的
opendir($dh, $somedir) || die "can't opendir $somedir: $!";
while (readdir($dh)) {
    print "$somedir/$_\n";
}
closedir $dh;
```

## readline



```
readline FILEHANDLE
readline
```

这是实现<FILEHANDLE>操作符的内部函数，不过也可以直接使用。这个函数从FILEHANDLE读下一条记录，这可能是一个文件句柄名，或者是一个间接文件句柄表达式，它可能返回真正文件句柄的名字，也可能返回一个引用，指向类似文件句柄对象的构造，如一个类型团。在标量上下文中，每个调用会读取并返回下一条记录，直到到达文件末尾，如果到达文件末尾，下一个调用会返回undef。在列表上下文中，readline会读取记录直到到达文件末尾，并返回一个记录列表。这里所说的“记录”通常是指一行文本，不过如果改变\$/ (\$INPUT\_RECORD\_SEPARATOR)的值，使它不再是其默认值，这会导致这个操作符以不同的方式对文本“分块”。取消\$/的定义会使块大小为整个文件（slurp模式）。

在标量上下文中“吞入”（slurp）文件时，如果正好吞入一个空文件，readline第一次会返回""，以后每次都会返回undef。从魔法ARGV文件句柄吞入时，每个文件会返回一个块（同样的，null文件会返回""），文件全部读取完时，最后会返回一个undef。如果省略FILEHANDLE，则假设处理ARGV文件句柄。

<FILEHANDLE>操作符在第2章“输入操作符”小节中有更详细的讨论。

```
$line = <STDIN>;
$line = readline(STDIN);           #作用相同
$line = readline(*STDIN);          #作用相同
$line = readline(\*STDIN);         #作用相同

open(my $fh, "<&=STDIN") || die;
bless($fh => "AnyOldClass");
$line = readline($fh);             #作用相同
```

## readlink

\$_	\$_	\$_	\$_
-----	-----	-----	-----

```
readlink EXPR
readlink
```

这个函数返回一个符号链接指向的文件名。*EXPR*要计算为一个文件名，它的最后一部分应当是一个符号链接。如果不是一个符号链接，或者如果符号链接在文件系统上未实现，或者出现了一些系统错误，将返回undef，应当检查\$!中的错误码。

注意返回的符号链接可能相对于你指定的位置。例如，你可能会写为：

```
$link_contents = readlink("/usr/local/src/express/yourself.h");
```

而readlink可能返回：

```
../express.1.23/includes/yourself.h
```

这并不能直接作为一个文件名使用，除非你的当前目录恰好是/usr/local/src/express。

## readpipe

\$_	\$_	\$_	\$_	\$_
-----	-----	-----	-----	-----

```
readpipe scalar EXPR
readpipe LIST # (建议)
```

这是实现qx//引号构造（也称为反引号操作符）的内部函数。有时你可能需要以某种方式指定*EXPR*，但如果使用引号形式可能不太方便，在这些情况下就可以使用qx//。注意我们将来可能会改变这个接口，来支持一个*LIST*参数，使它更像exec函数，所以不要假设它还会继续为*EXPR*提供标量上下文。你要自己通过*scalar*指定标量上下文，或者尝试*LIST*形式。谁知道呢，可能等你读到这个内容时*LIST*形式已经可以用了。

## recv

\$_	\$_	\$_	\$_	\$_
-----	-----	-----	-----	-----

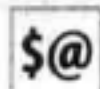
```
recv SOCKET, SCALAR, LEN, FLAGS
```

这个函数接收套接字上的一个消息。它试图从指定的*SOCKET*文件句柄接收*LENGTH*个字符（码点）的数据，并放入变量*SCALAR*中。这个函数返回发送者的地址，或者如果出现一个

错误则返回undef。SCALAR会扩展或收缩到实际读取的长度。这个函数与recv(2)接收相同的标志，实际上它就是使用recvfrom(2)实现的。参见第15章“套接字”一节。

注意这里所说的“字符”：取决于套接字的状态，可能接收未编码的（8位）字节，或者接收完全解码的字符。默认地，所有套接字都处理字节。不过，举例来说，如果使用binmode改变套接字时提供了:encoding(utf8) I/O层，I/O就会处理UTF-8编码的Unicode字符，而不是字节。

## redo



```
redo LABEL
redo
```

redo操作符重启一个循环块，但不会重新计算条件。如果有continue块，不会执行这个块。如果省略LABEL，这个操作符指示最内层外围循环。正常情况下，如果程序想骗自己说这就是输入，就可以使用这个操作符：

```
# 这个循环用一个反斜线连接后续的行
while (<STDIN>) {
    if (s/\\n$// && defined($nextline = <STDIN>)) {
        $_ .= $nextline;
        redo;
    }
    print; #或其他...
}
```

redo不能用来退出一个有返回值的块，如eval {}、sub{}或do {}，另外不要用redo来退出一个grep或map操作。如果启用了警告，使用redo退出一个不在当前词法作用域中的循环时，Perl会发出警告。

块本身在语义上等同于只执行一次的循环。因此，这样一个块中的redo实际上会把它变成一个循环构造。参见第4章“循环控制”一节。

## ref



```
ref EXPR
ref
```

如果EXPR是一个引用，ref操作符返回true，否则返回false。返回的值取决于引用所指示的对象的类型。内置类型包括：

```
SCALAR
ARRAY
HASH
CODE
REF
```



GLOB  
LVALUE  
FORMAT  
IO  
VSTRING  
Regexp

返回值LVALUE指示一个非变量的左值的引用。如果取函数调用（如pos或substr）的引用就会得到这个返回值。如果引用指向一个版本串，则返回VSTRING。

结果Regexp指示参数是从qr//得到的一个正则表达式。

如果引用的对象已经祝福到一个包，则返回这个包的包名。可以认为ref一个“typeof”（取类型）操作符。

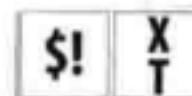
```
if (ref($r) eq "HASH") {  
    say "r is a reference to a hash.";  
}  
elsif (ref($r) eq "Hump") { # 调皮，见下文  
    say "r is a reference to a Hump object.";  
}  
elsif (not ref $r) {  
    say "r is not a reference at all.";  
}
```

如果测试你的对象类是否与某个特定类的类名相同，通常认为这是一种不好的OO风格，因为派生类可能有一个不同的名字，但完全可以访问基类的方法（根据Liskov替换原则）。更好的做法是使用UNIVERSAL方法isa，如下所示：

```
if ($r->isa("Hump")) {  
    say "r is a reference to a Hump object, || subclass.";  
}
```

通常最好根本不做测试，因为OO机制不会向你的方法发送对象，除非它认为这样合适。有关的更多详细信息参见第8章和第12章。另外参见第29章“attributes”一节中的reftype函数。

## rename



rename OLDNAME, NEWNAME

这个函数用来改变一个文件的名字。成功时返回true，否则返回false。这个函数（通常）不能跨文件系统使用，不过在一个Unix系统上，有时可以使用mv命令来弥补这一点。如果一个名为NEWNAME的文件已经存在，它将被删除。非Unix系统可能还有另外的限制。

参见标准File::Copy模块，可以了解如何使用一个平台独立的move函数实现跨文件系统重命名。

## require

```
require VERSION
require EXPR
require
```

这个函数断言对其参数的某种依赖性。

如果参数是一个字符串，这个字符串给定了一个文件的名称，`require`会加载并执行这个文件中找到的Perl代码。这类似于在一个文件上使用`do`，只不过`require`会查看是否已经加载这个库文件，如果遇到任何问题，会产生一个异常（因此可以用它来表述文件依赖关系，而不用担心重复编译）。与相应的`do`和`use`类似，`require`知道如何搜索存储在`@INC`数组中的路径，成功时会更新`%INC`。参见第25章。

文件必须返回`true`作为最后一个值，以指示成功地执行了一些初始化代码，所以我们通常会用一个`1`结束这样一个文件，除非你确信文件会返回`true`（这个要求将来可能会放宽）。

如果`require`的参数是一个v5.6.2形式的版本号，`require`则要求当前执行的Perl版本至少是这个版本（Perl还可以接收一个浮点数，如5.005\_03，从而与老版本的Perl兼容，不过这种形式现在已经不建议使用，因为来自其他文化的人无法理解这种形式）。因此，如果一个脚本需要v5.14版本，可以把以下任意一个声明作为脚本的第一行：

```
require 5.014_001;    # 用于（与老版本）向后兼容
require 5.14.1;       # 同上
require v5.14.1;      # 运行时版本检查
```

老版本的Perl会退出。不过，与所有`require`类似，这是在运行时完成的。你可能希望使用`use 5.14.0`来完成编译时检查。参见第25章中的`$PERL_VERSION`。

如果`require`的参数是一个裸包名（参见`package`），`require`会假设有一个自动的`.pm`后缀，使它很容易加载标准模块。这种行为类似于`use`，只不过它发生在运行时而不是编译时，而且不会调用`import`方法。例如，要加入`Socket.pm`但不向当前包中引入任何符号，可以写为：

```
require Socket; # 不是"use Socket;"
```

不过，用以下声明可以得到同样的效果，它还有一个优点，如果没有找到`Socket.pm`，会给出一个编译时警告：

```
use Socket ();
```

在一个裸名上使用`require`还会把包名中的所有`::`替换为你的系统的目录分隔符，一般是`/`。换句话说，如果做以下声明：

```
require Foo::Bar;    # 一个漂亮的裸名
```

`require`函数会在`@INC`数组指定的目录中查找`Foo/Bar.pm`文件。不过，如果使用以下代码：

```
$class = "Foo::Bar";
require $class;      # $class不是一个裸名
```

或者以下代码：

```
require "Foo::Bar";  # 加引号的直接量不是一个裸名
```

`require`函数会在`@INC`数组中查找`Foo::Bar`文件，如果在那里没有找到`Foo::Bar`，会做出警告。如果是这样，可以执行以下代码：

```
eval "require $class";
```

现在你已经了解了`require`如何用一个裸字参数查找文件，它在后台还有另外一个小功能。`Require`查找“.pm”扩展名之前，首先会查找一个有“.pmc”扩展名的类似文件名。如果找到这个文件，则会加载这个文件，并取代以“.pm”扩展名结尾的任何文件。

`@INC`数组包含一个标量列表，这些标量确实了如何加载一个模块。`require`函数会遍历这个列表，直到找到一个标量记录从而能得到可加载的源代码，然后加载这个代码。

`@INC`的各个元素可以是一个字符串（这会看作是一个目录名，将在这个目录中查找要加载的文件），或者是某种形式的“类代码实体”（用来生成所加载文件的内容）。

“类代码实体”可以是一个子例程引用、一个包含子例程引用的数组（以及子例程的其他可选的参数），或者是有一个`INC`方法的对象。不论遇到什么形式的“类代码实体”，都会调用这个代码，并传入两个参数：实体本身和所查找的文件。如下所示：

```
Sub ref: $sub_ref->($sub_ref, $required_file)
Array ref: $arr_ref->[0]->($arr_ref, $required_file)
Object: $object->INC($required_file)
```

不论调用哪一种形式，子例程或方法都总是要返回一个至少包含3个值的列表，其解释如表27-4所示。

表27-4：@INC中代码引用期望的返回值

参数	动作
(HANDLE)	从这个句柄读入源代码
(HANDLE, CODEREF)	从这个句柄读入源代码，并通过子例程过滤
(HANDLE, CODEREF, REF)	与上面相同，不过还会向子例程传入REF
(undef, CODEREF)	反复调用子例程来返回源代码行
(undef, CODEREF, REF)	与上面相同，不过还会向子例程传入REF
其他	失败，并尝试@INC中的下一项



这些钩子还允许对应其加载的文件设置%INC项。参见第25章中的%INC。

另外参见do *FILE*、use命令、lib pragma和标准FindBin模块。

## reset

```
reset EXPR
reset
```

这个函数通常在循环最上面使用（或滥用），或者在循环末尾的一个continue块中使用，来清除全局变量或重置m??搜索，使它们重新开始工作。表达式会解释为一个单字符列表（允许用连字符指示范围）。所有以其中某个字母开头的标量变量、数组和散列都重置为其最初的状态。如果省略表达式，单次匹配搜索（m?PATTERN?）会重置，重新进行匹配。这个函数只重置当前包的变量或搜索。它总是返回true。

要重置所有“X”变量，可以写为：

```
reset "X";
```

要重置所有小写变量，可以写为：

```
reset "a-z";
```

最后，如果只重置??搜索，可以写为：

```
reset;
```

不建议重置包main中的“A-Z”，因为这样一来，你会清除你的全局ARGV、INC、ENV以及SIG数组和散列。

由my创建的词法作用域变量不受影响。reset基本上已经废弃，因为它很容易将整个命名空间都清除，另外由于??操作符本身也基本上已经废弃，请改为使用m??。

参见标准Symbol模块的delete\_package函数，另外关于安全隔离室的问题在第20章“安全隔离室”一节有详细介绍。

## return

```
return EXPR
return
```

这个操作符导致当前子例程、eval或do文件以指定的值（也可能有多个值）立即返回。如果试图在这3个地方以外使用return，会产生一个异常。还要注意，eval不能代表调用这个eval的子例程执行return。

EXPR将在列表上下文、标量上下文或void上下文计算，这取决于如何使用返回值，而且每



次执行时这一点可能会改变。也就是说，提供的表达式会在调用这个子例程的相同的上下文中计算。如果子例程在标量上下文中调用，*EXPR*也在标量上下文中计算，所以会返回一个标量值。如果子例程在列表上下文中调用，*EXPR*也在列表上下文中计算，因此会返回一个值列表。无参数的`return`在标量上下文中返回标量值`undef`，在列表上下文中返回一个空列表`()`，在`void`上下文中（很自然地）什么也不返回。子例程调用的上下文可以在子例程中使用`wantarray`函数来确定（这个函数的名字并不贴切）。

## reverse

`reverse LIST`

在列表上下文中，这个函数返回一个列表值，由*LIST*中的元素组成，但顺序相反。这个函数可以用来创建降序序列：

```
for (reverse 1 .. 10) { ... }
```

作为一个*LIST*传递时，由于散列会扁平化为列表，所以`reverse`也可以用来逆转散列（假设值是唯一的）：

```
%barfoo = reverse %foobar;
```

在标量上下文中，这个函数将*LIST*的所有元素连接在一起，并逐字符地返回所得到的字符串的逆串。这里的字符是指码点，而不是字形。这说明，有可能将“`\r\n`”错误地逆转为“`\n\r`”，导致不正确地对基字符应用组合字符。要想按字形而不是按码点逆转，可以这样做：

```
$codeuni = join "" => reverse $unicode =~ /\X/g;
```

一个小提示：如果要逆转通过一个用户自定义函数排序的列表，更容易的做法是一开始就采用相反的方向对这个列表排序。

## rewinddir



`rewinddir DIRHANDLE`

这个函数为读取*DIRHANDLE*的`readdir`例程将当前位置设置为目录的起始位置。并不是支持`readdir`的所有机器都支持这个函数。如果未实现，`rewinddir`会退出。成功时返回`true`，否则返回`false`。

## rindex

```
rindex STR, SUBSTR, POSITION  
rindex STR, SUBSTR
```

这个函数的工作类似于`index`，不过它返回`STR`中`SUBSTR`最后一次出现的位置（反向`index`）。如果没有找到`SUBSTR`，这个函数返回-1。如果指定了`POSITION`，这是可能返回的最右位置。

要反向遍历一个字符串，可以写为：

```
$pos = length $string;
while (($pos = rindex $string, $lookfor, $pos) >= 0) {
    say "Found at $pos";
    $pos--;
}
```

需要说明，类似于`index`，这会按字符（码点）位置处理，而不是按字形位置。要处理作为字形序列而不是码点序列的字符串，参见CPAN `Unicode::GCString`模块的`index`、`rindex`和`pos`方法。

## rmdir

\$_	\$_!	X
		T

```
rmdir FILENAME
rmdir
```

这个函数会删除`FILENAME`指定的目录（如果该目录为空）。函数成功时返回`true`，否则返回`false`。如果你想先删除目录的内容，但是出于某种原因不能通过shell调用`rm -r`（如没有一个shell或者没有`rm`命令），可以参见`File::Path`模块。

## s///

T	X	X
	ARG	T

```
s///
```

替换操作符。参见第5章中“模式匹配操作符”一节。

## say

\$_	\$_!	X
		ARG

```
say FILEHANDLE LIST
say FILEHANDLE
say LIST
say
```

类似于`print`，不过会隐式地追加一个换行符。`say LIST`就是`{ local $_ = "\n"; print LIST }`的一个缩写形式。如果要使用`FILEHANDLE`，但是没有一个`LIST`可以向其中打印`$_`的内容，则必须使用一个真正的文件句柄，如`FH`，而不能是一个间接文件句柄，如`$fh`。

这个关键字只在“say”特性启用时可用，参见第3章“项和列表操作符（左边）”一节。



## scalar

scalar *EXPR*

这个伪函数可以在`LIST`中用来强制`EXPR`在标量上下文中计算（如果在列表上下文中计算会生成一个不同的结果）。

例如：

```
my($nextvar) = scalar <STDIN>;
```

在赋值之前会阻止`<STDIN>`从标准输入读取所有行，因为对列表的赋值（甚至一个`my`列表）会提供列表上下文。在这个例子中，如果没有`scalar`，`<STDIN>`中的第一行仍赋值给`$nextvar`，但是后面的行读取后就会被丢掉，因为我们要赋值的目标列表只能接收一个标量值。当然，一个更简单的方法（而且不那么混乱）是去掉小括号，这样就把列表上下文变成了一个标量上下文：

```
my $nextvar = <STDIN>;
```

由于`print`函数是一个`LIST`操作符，如果你希望打印出`@ARRAY`的长度，必须写为：

```
say "Length is ", scalar(@ARRAY);
```

并没有一个与`scalar`对应的“`list`”函数，因为实际上根本不需要强制在列表上下文中计算。这是因为，只要操作要接收`LIST`，所有这些操作都已经为其列表参数免费提供了列表上下文。

由于`scalar`是一个一元操作符，如果你不小心对`EXPR`使用了一个有小括号的列表，这会表现为一个标量逗号表达式，在`void`上下文中只会计算最后一个元素，而在标量上下文中会返回计算的最后一个元素。这往往不是你想要的。下面这个语句：

```
print uc(scalar(&foo,$bar)),$baz;
```

基本上等价于以下的两个语句：

```
&foo;  
print(uc($bar),$baz);
```

关于逗号操作符的更多详细信息参见第2章，另外对于一元操作符参见第7章中“原型”一节。

## seek



seek *FILEHANDLE*, *OFFSET*, *WHENCE*

这个函数定位`FILEHANDLE`的文件指针，类似于标准I/O的`fseek(3)`调用。文件起始位置的偏移量为0，而不是偏移量为1。另外，偏移量是指字节位置，而不是字符位置或行号。一般

地，由于行长度会变化，所以如果要访问某个特定的行号，需要检查从起点到该点的所有文件内容，否则无法访问，除非所有行都已知为某个特定的长度，或者你已经建立了一个索引，可以将行号转换为字符偏移量。（如果文件采用变长字符编码（如UTF-8，UTF-16），那么对于文件中的字符位置也有同样的限制，操作系统不知道什么是字符，它只知道字节）。

*FILEHANDLE*可以是一个表达式，这个表达式的值给出一个实际文件句柄、类型团或类文件句柄对象的名字。这个函数成功时返回true，否则返回false。为方便起见，这个函数可以从不同的文件位置为你计算偏移量。*WHENCE*的值指定*OFFSET*使用哪个文件位置作为其起点：0表示文件起始位置；1为文件的当前位置；或者2为文件的末尾。如果*WHENCE*为1或2，*OFFSET*可以为负。如果你想对*WHENCE*使用一个符号值，可以使用IO::Seekable或者POSIX模块或Fcntl模块的SEEK\_SET、SEEK\_CUR和SEEK\_END。

如果你想定位文件来执行sysread或syswrite,不要使用；由于有标准I/O缓冲，seek对于文件的系统位置的影响是不可预测的，而且这个函数不可移植。应当使用sysseek。

基于ANSI C的规则和严格性，在一些系统上，读和写之间切换时必须完成一个seek定位。这会调用标准I/O库的clearerr(3)函数，另外还有其他一些效果。如果*WHENCE*为1 (SEEK\_CUR)而且*OFFSET*为0，这样可以不移动文件位置：

```
seek(TEST, 0, 1);
```

这个函数还有一个有趣的用法，允许你追随不断扩大的文件，如下：

```
for (;;) {
    while (<LOG>) {
        grok($_);          # 处理当前行
    }
    sleep 15;
    seek LOG, 0, 1;        # 重置文件末尾错误
}
```

最后一个seek会清除文件末尾错误，而不会移动文件指针。取决于你的C库的标准 I/O实现有多标准，你可能需要类似下面的代码：

```
for (;;) {
    for ($curpos = tell FILE; <FILE>; $curpos = tell FILE) {
        grok($_);          # 处理当前行
    }
    sleep $for_a_while;
    seek FILE, $curpos, 0; # 重置文件末尾错误
}
```

可以用类似的策略将各行的seek地址记在一个数组中。

警告：不论文件句柄上是否有编码层，*POSITION*都按字节而不是字符定位。不过，Perl中

所有读文件的函数都要经过某个编码层，因此你可能读到一个不完整的部分“字符”，并形成一个不合法的Perl字符串。在有多字节编码层的文件句柄上，要避免混合使用sysseek或seek调用和I/O函数。

## seekdir



`seekdir DIRHANDLE, POS`

这个函数为下一个读取DIRHANDLE的readdir调用设置当前位置。POS必须是telldir返回的一个值。这个函数与相应的系统库例程在可能的目录压缩方面存在同样的问题。并不是实现了readdir的所有平台上都实现了这个函数，当然没有实现readdir的平台更不会实现这个函数。

## select（输出文件句柄）



`select FILEHANDLE`  
`select`

由于历史的原因，有两个select操作符，它们相互之间没有任何关系（另一个select操作符见下一节）。这个版本的select操作符返回当前选择的输出文件句柄，如果提供了FILEHANDLE，则设置当前默认的输出文件句柄。这有两个作用：首先，没有提供文件句柄的write或print会默认使用这个FILEHANDLE；其次，与输出相关的特殊变量会指示这个输出文件句柄。例如，如果你要为多个输出文件句柄设置表格顶端格式，可以这样做：

```
select REPORT1;
$^ = "MyTop";
select REPORT2;
$^ = "MyTop";
```

不过，注意这会保留REPORT2作为当前选择的文件句柄。这简直是反社会的，因为它实际上会搞乱其他例程的print或write语句。编写得当的库例程会保证其退出和进入时当前选择的文件句柄是一样的。为支持这一点，FILEHANDLE可以是一个表达式，这个表达式的值给出实际文件句柄的名字。因此，可以如下保存和恢复当前选择的文件句柄：

```
my $oldfh = select STDERR;
$| = 1;
select $oldfh;
```

或者采用常用但有些含糊的做法，如下所示：

```
select((select(STDERR), $| = 1)[0])
```

这个例子会建立一个列表，包含select(STDERR)的返回值（作为一个副作用，它会选择STDERR）和\$| = 1（这总为1），另外还在当前选择的STDERR上设置自动刷新输出，这也



作为一个副作用。这个列表的第一个元素（之前选择的文件句柄）现在用作为外部select的参数。很奇怪，是吧？由此你应该知道Lisp有多危险了。

还可以使用标准SelectSaver模块在作用域退出时自动恢复之前的select。

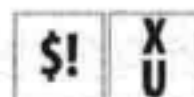
不过，虽然我们已经解释了所有这些内容，但应当指出，如今已经很少需要使用这种形式的select，因为对于你要设置的大多数特殊变量，都已经有了相应的面向对象包装器方法可以为你完成这个工作。所以不用直接设置\$I，可以写为：

```
use IO::Handle;                # 很遗憾，这*不*是一个小模块
STDOUT->autoflush(1);
```

之前的格式例子可以写为：

```
use IO::Handle;
REPORT1->format_top_name("MyTop");
REPORT2->format_top_name("MyTop");
```

## select（准备好的文件描述符）



```
select RBITS, WBITS, EBITS, TIMEOUT
```

4参数的select操作符与之前介绍的select操作符没有任何关系。这个操作符用于发现你的哪一个文件描述符（如果有）已经准备就绪可以完成输入或输出，或者报告一个异常条件（这有助于避免轮询）。它会利用你指定的位掩码调用select(2)系统调用，位掩码可以使用fileno和vec构造，如下所示：

```
$rin = $win = $ein = "";
vec($rin, fileno(STDIN), 1) = 1;
vec($win, fileno(STDOUT), 1) = 1;
$ein = $rin | $win;
```

如果你想对多个文件句柄使用select，可以写一个子例程：

```
sub fhbits {
    my @fhlist = @_;
    my $bits;
    for my $fh (@fhlist) {
        vec($bits, fileno($fh), 1) = 1;
    }
    return $bits;
}
$rin = fhbits(*STDIN, *TTY, *MYSOCK);
```

注意我们使用了类型团来传入这些文件句柄，因为按字符串传入文件句柄是个糟糕的想法。如果使用自动生成的文件句柄，则不用这么做。

如果想反复使用相同的位掩码（而且这样会更高效），通常的做法是：

```
($nfound, $timeleft) =  
    select($rout=$rin, $wout=$win, $eout=$ein, $timeout);
```

或者阻塞直到有文件描述符准备就绪：

```
$nfound = select($rout=$rin, $wout=$win, $eout=$ein, undef);
```

可以看到，在标量上下文中调用select只会返回\$nfound，即找到的已准备就绪的描述符的个数。

之所以可以使用\$wout=\$win技巧，这是因为赋值的值在它的左边，所以\$wout先由赋值修改，再被select修改，而\$win保持不变。

任何一个参数都可以为undef，在这种情况下，它们会被忽略。TIMEOUT如果不为undef，这可能是一个小数，单位为秒（超时时间为0会影响轮询）。并没有太多实现能够返回\$timeleft。如果做不到，则总会将你提供的\$timeout作为\$timeleft返回。

标准IO::Select模块为select提供了一个对用户更友好的接口，主要是因为它会为你完成所有位掩码工作。

select的一种用法是细粒度地控制睡眠，比sleep所允许的粒度更细。为此，要为所有位掩码指定undef。所以要睡眠（至少）4.75秒，可以使用：

```
select undef, undef, undef, 4.75;
```

在一些非Unix系统上，不能连用3个undef，可能至少需要为一个合法的描述符假造一个位掩码，尽管它永远也不会准备就绪。

如今，要完成这一点，更可移植的方法是从标准Time::HiRes模块导入一个特殊版本的sleep：

```
use Time::HiRes qw(sleep);  
sleep 4.75;    # 不是正常的睡眠
```

不要（尝试）将缓冲I/O操作（如read或<HANDLE>）与select混合使用，除非POSIX允许，即使如此也只能在真正的POSIX系统上使用。更合适的做法是使用sysread。

## semctl

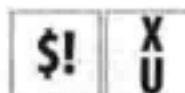


```
semctl ID, SEMNUM, CMD, ARG
```

这个函数调用System V IPC函数semctl(2)。你可能先要声明use IPC::SysV才能得到正确的常量定义。如果CMD是IPC\_STAT或GETALL，那么ARG必须是一个变量，包含返回的semid\_ds结构或信号量值数组。返回值与ioctl和fcntl类似：undef表示错误，“0 but true”表示0，或者是具体的返回值。

参见IPC::Semaphore模块。这个函数只在支持System V IPC的机器上可用。

## semget

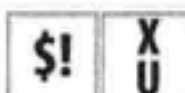


`semget KEY, NSEMS, FLAGS`

这个函数调用System V IPC系统调用`semget(2)`。调用之前，应当声明`use IPC::SysV`来得到正确的常量定义。这个函数返回信号量ID，或者如果存在一个错误则返回`undef`。

参见IPC::Semaphore模块。这个函数只在支持System V IPC的机器上可用。

## semop



`semop KEY, OPSTRING`

这个函数调用System V IPC系统调用`semop(2)`来完成信号量操作，如发出信号和等待。调用之前，应当声明`use IPC::SysV`来得到正确的常量定义。

`OPSTRING`必须是`semop`结构的一个打包数组。可以用`pack("s*", $semnum, $semop, $semflag)`来建立各个`semop`结构。信号量操作的数目由`OPSTRING`的长度指示。这个函数成功时返回`true`，否则如果存在一个错误则返回`false`。

下面的代码要等待信号量`$semnum`（信号量id为`$semid`）：

```
$semop = pack "s*", $semnum, -1, 0;  
semop($semid, $semop) || die "Semaphore trouble: $!";
```

要发出这个信号量，只需把-1替换为1。

参见第15章“System V IPC”一节。另外参见IPC::Semaphore模块。这个函数只在支持System V IPC的机器上可用。

## send



`send SOCKET, MSG, FLAGS, TO`  
`send SOCKET, MSG, FLAGS`

这个函数在套接字上发送一个消息。它与同名的系统调用有相同的标志，参见`send(2)`。在未连接的套接字上，必须指定一个要发送消息的目标`TO`，这使得Perl的`send`的工作与`sendto(2)`类似。目前标准Perl中还没有实现C系统调用`sendmsg(2)`。`send`函数会返回发送的字符个数，或者如果存在一个错误则返回`undef`。

注意这里所说的“字符”：取决于套接字的状态，可能接收未编码的（8位）字节，或者接收完全解码的字符。默认地，所有套接字都处理字节。不过，举例来说，如果使用



binmode改变套接字时提供了:encoding(utf8) I/O层, I/O就会处理UTF-8编码的Unicode字符, 而不是字节。

一些非Unix系统错误地将套接字与普通的文件描述符区别处理, 这样一来, 必须在套接字上使用send和recv, 而不能使用更方便的标准I/O操作符。

我们可能经常犯一个错误, 把Perl的send和C的send混淆在一起, 而写出这样的代码:

```
send SOCK, $buffer, length $buffer;    #不正确
```

这会莫名其妙地失败, 具体情况取决于字符串长度与系统期望的`FLAGS`位之间的关系。有关的例子参见第15章“消息传递”一节。

## setpgrp



```
setpgrp PID, PGRP
```

这个函数为指定的`PID` (当前进程使用`PID 0`)设置当前进程组 (`PGRP`)。如果系统没有实现`setpgrp(2)`, 调用`setpgrp`会产生一个异常。注意: 有些系统会忽略你提供的参数, 总是执行`setpgrp(0, $$)`。幸运的是, 这些正是我们通常想要提供的参数。如果省略了参数, 就默认为`0,0`。BSD 4.2版本的`setpgrp`不接收任何参数, 不过在BSD 4.4中, 它是`setpgid`函数的一个同义词。为了提供更好的可移植性 (根据定义), 可以直接使用POSIX模块中的`setpgid`函数。如果你只是想让你的脚本成为守护进程, 也可以考虑使用`POSIX::setsid`函数。需要说明, POSIX版本的`setpgrp`不接收任何参数, 所以只有`setpgrp(0,0)`是真正可移植的。

## setpriority



```
setpriority WHICH, WHO, PRIORITY
```

这个函数为`WHICH`和`WHO`指定的进程、进程组或用户设置当前`PRIORITY`。参见`setpriority(2)`。在未实现`setpriority(2)`的机器上调用`setpriority`会产生一个异常。要将进程的优先级“调整”4个单位 (等同于用`nice(1)`执行你的程序), 可以尝试:

```
setpriority 0, 0, getpriority(0, 0) + 4;
```

对于一个给定优先级的解释可能会随不同的操作系统而有所不同。有些优先级对于非特权用户可能不可用。参见CPAN的`BSD::Resource`模块。

## setsockopt



```
setsockopt SOCKET, LEVEL, OPTNAME, OPTVAL
```

这个函数设置所请求的套接字选项。如果有错误，这个函数返回undef。Socket模块为LEVEL和OPNAME提供了所需的常量，不过LEVEL的常量都可以从getprotobyname得到。LEVEL指定调用目标是哪个协议层，或者可以是SOL\_SOCKET，表示在所有层之上的套接字本身。OPTVAL可以是一个打包的字符串或者是一个整数。整数OPTVAL是pack("i", OPTVAL)的一个简写。如果不希望传入参数，可以指定OPTVAL为undef。

通常会在套接字上设置选项SO\_REUSEADDR，它会绕过这样一个问题：如果一个端口上前一个TCP连接还在下决心关闭连接，将无法绑定一个特定的地址。如下所示：

```
use Socket;
socket(SOCK, ...) || die "Can't make socket: $!";
setsockopt(SOCK, SOL_SOCKET, SO_REUSEADDR, 1)
    || warn "Can't do setsockopt: $!\n";
```

另一个常见的选项是在套接字上禁用Nagle算法：

```
use Socket qw(IPPROTO_TCP TCP_NODELAY);
setsockopt($socket, IPPROTO_TCP, TCP_NODELAY, 1);
```

其他的可取值参见setsockopt(2)。

## shift



```
shift ARRAY
shift
```

这个函数移除数组中的第一个值，并返回这个值，将数组长度减1，同时让所有其余元素前移（或者上移或左移，取决于你如何查看数组列表。我们喜欢说左移）。如果数组中没有元素，这个函数返回undef。

如果省略ARRAY，函数在子例程和格式的词法作用域中会对@\_完成shift操作；在文件作用域（通常是主程序）或者eval STRING、BEGIN {}、CHECK {}、UNITCHECK{}、INIT {}和END {}构造建立的词法作用域中会对@ARGV完成shift操作。

子例程通常首先将其参数复制到词法作用域变量，可以shift用来完成这个工作：

```
sub marine {
    my $fathoms = shift;      # 深度
    my $fishies = shift;     # 鱼数量
    my $o2 = shift;          # 氧含量
    # ...
}
```

shift还可以用在程序最前面来处理参数：

```
while (defined($_ = shift)) {
    /^[^-]/      && do { unshift @ARGV, $_; last };
    /^-w/        && do { $WARN = 1;      next };
}
```

```

    /^-r/      && do { $RECURSE = 1;      next };
    die "Unknown argument $_";
}

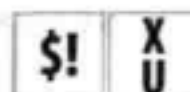
```

要处理程序参数，应当考虑标准`Getopt::Std`和`Getopt::Long`模块。

从v5.14开始，`shift`可以取一个未祝福的数组的引用，它会自动解引用。`shift`的这个方面还是试验性的。具体的行为在Perl将来的版本中有可能改变。

参见`unshift`，`push`，`pop`和`splice`。`shift`和`unshift`函数在数组左边与`pop`和`push`在数组右边做的工作相同。

## shmctl



`shmctl ID, CMD, ARG`

这个函数调用System V IPC系统调用`shmctl(2)`。调用之前，应当声明`use IPC::SysV`来得到正确的常量定义。

如果`MD`是`IPC_STAT`，那么`ARG`必须是一个变量，包含返回的`shmid_ds`结构。返回值与`ioctl`和`fcntl`类似：`undef`表示错误，“0 but true”表示0，或者是具体的返回值。

这个函数只在支持System V IPC的机器上可用。

## shmget

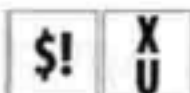


`shmget KEY, SIZE, FLAGS`

这个函数调用System V IPC系统调用`shmget(2)`。该函数返回共享内存段ID，或者如果存在一个错误则返回`undef`。调用之前，应当声明`use SysV::IPC`。

这个函数只在支持System V IPC的机器上可用。

## shmread



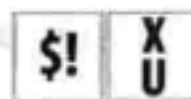
`shmread ID, VAR, POS, SIZE`

这个函数从共享内存段`ID`读取数据，从位置`POS`开始，大小为`SIZE`（需要关联共享内存段、复制出数据，再与内存段解除关联）。`VAR`是将包含所读数据的变量。这个函数成功时返回`true`，如果出现错误则返回`false`。

这个函数只在支持System V IPC的机器上可用。



## shmwrite



shmwrite *ID*, *STRING*, *POS*, *SIZE*

这个函数写至共享内存段*ID*，从位置*POS*开始，大小为*SIZE*（需要关联共享内存段、向其中复制数据，再解除关联）。如果字符串太长，那么只写入*SIZE*个字节；如果字符串太短，会写入null填满*SIZE*个字节。这个函数成功时返回true，如果出现错误则返回false。

这个函数只在支持System V IPC的机器上可用（这句话你可能已经看烦了，说实在的，我们也说烦了）。

## shutdown



shutdown *SOCKET*, *HOW*

这个函数会以*HOW*指示的方式关闭一个套接字连接。如果*HOW*为0，则禁止进一步接收数据。如果*HOW*为1，则禁止继续发送数据。如果*HOW*为2，所有操作都禁止。

```
shutdown(SOCK, 0);    # 不允许继续读
shutdown(SOCK, 1);    # 不允许继续写
shutdown(SOCK, 2);    # 不允许任何I/O
```

如果你想告诉套接字另一端你已经完成了写但是还没有完成读，或者反之，这个函数就很有用。这也是一种更坚定的关闭方式，因为它会禁用派生子进程中保存的这些文件描述符的所有副本。

假设一个服务器想读取客户端的请求，直到文件末尾，然后发回一个答案。如果客户端调用了close，现在这个套接字就不能完成I/O，所以不会发回任何答案。因此客户应当使用shutdown将这个连接“半关”：

```
say SERVER "my request";    # 发送一些数据
shutdown(SERVER, 1);        # 发送eof；不允许继续写
$answer = <SERVER>;         # 不过仍可以读
```

如果你想知道如何关闭你的系统，可以执行一个外部程序来做到。参见system。

## sin



sin *EXPR*  
sin

抱歉，这个操作符实在没有什么过错<sup>译者注</sup>。它只是返回*EXPR*（用弧度表示）的正弦。

要完成反正弦操作，可以使用Math::Trig或POSIX模块的asin函数，或者使用以下关系：

---

译者注：sin在英语中有罪恶的意思。

```
sub asin { atan2($_[0], sqrt(1 - $_[0] * $_[0])) }
```

## sleep

```
sleep EXPR
sleep
```

这个函数会让脚本睡眠*EXPR*（整数）秒，或者如果没有*EXPR*则永远睡眠，并返回睡眠的秒数。可以通过向进程发送一个SIGALRM来中断睡眠。在一些较老的系统上，它可能会比你请求的秒数少整整一秒，这取决于它如何计算时间。大多数现代系统都会睡足量。不过，看上去睡眠时间可能比指定的时间长，因为如果在一个很忙的多任务系统中，可能没有调度你的进程立即执行。对于更细粒度的延迟（小于1秒），标准Time::HiRes模块提供了一个usleep函数。如果可以使用，select（准备好的文件描述符）调用也能提供更细的粒度。可以使用syscall来调用一些Unix系统支持的getitimer(2)和setitimer(2)例程。alarm和sleep调用不能混合使用，因为sleep通常就是用alarm实现的。

参见POSIX模块的pause函数。

## socket

\$!	X	X	X
	ARG	T	U

```
socket SOCKET, DOMAIN, TYPE, PROTOCOL
```

这个函数会打开指定类型的套接字，并把它关联到文件句柄*SOCKET*。*DOMAIN*、*TYPE*和*PROTOCOL*的指定与socket(2)相同。如果未定义，*SOCKET*会自动生成。使用这个函数之前，程序应当包含下面这行代码：

```
use Socket;
```

这会提供适当的常量。如果成功，这个函数返回true。参见第15章“套接字”一节中的例子。

在支持“执行时关闭”（close-on-exec）标志的系统上，可以为新打开的文件描述符设置这个标志，由\$^F的值确定。另外参见第25章中的\$^F(\$SYSTEM\_FD\_MAX)变量。

## socketpair

\$!	X	X	X
	ARG	T	U

```
socketpair SOCKET1, SOCKET2, DOMAIN, TYPE, PROTOCOL
```

这个函数在指定类型的指定域中创建一对匿名的套接字。*DOMAIN*、*TYPE*和*PROTOCOL*的指定与socketpair(2)相同。可能需要声明use Socket来得到所需的常量。如果所有套接字参数都未定义，它们会自动生成。这个函数成功时返回true，否则返回false。在未实现socketpair(2)的系统上，调用这个函数会产生一个异常。

这个函数通常在fork前使用。得到的一个进程要关闭SOCKET1，另一个进程关闭SOCKET2。可以双向使用这些套接字，这与pipe函数创建的文件句柄不同。有些系统使用socketpair定义pipe，在这些系统上，pipe(Rdr, Wtr)调用实际上等同于：

```
use Socket;
socketpair(Rdr, Wtr, AF_UNIX, SOCK_STREAM, PF_UNSPEC);
shutdown(Rdr, 1);      # 不再向读者写
shutdown(Wtr, 0);      # 不再从写者读
```

在Perl v5.8及以后的版本中，如果系统实现了套接字但是没有实现socketpair，则使用主机的IP套接字来模拟socketpair。在支持“执行时关闭”（close-on-exec）标志的系统上，可以为新打开的文件描述符设置这个标志，由\$^F的值确定。参见第25章中的\$^F(\$SYSTEM\_FD\_MAX)变量。另外参见第15章“双向通信”一节最后的例子。

## sort

\$@

```
sort USERSUB LIST
sort BLOCK LIST
sort LIST
```

这个函数对LIST排序，并返回排序后的列表值。未定义的值会排在已定义的null串前面，null串则排在所有其他内容前面。默认地，它会按简单的数值码点顺序排序（或者如果重载了cmp操作符，会按cmp操作符返回的结果排序）。要完成真正的词典顺序排序，必须使用Unicode::Collate模块，参见第6章中的“Unicode文本比较和排序”。简单地讲，要得到一个好的字母顺序排序，最容易的方法如下：

```
use Unicode::Collate;
@alphabetized_list = Unicode::Collate->new->sort(@list);
```

locale pragma起作用时，sort LIST会根据当前的排序本地化环境对LIST排序。即使有这样一个本地化环境，然而Perl并不支持多字节本地化环境，所以可能与你预想的做法不同。如果想要实现可靠的本地化环境排序，可以参见Unicode::Collate::Locale模块。

如果给出了USERSUB，这是一个子例程名，它会根据如何对列表中的元素排序返回一个小于、等于或大于0的整数（还有很方便的<=>和cmp操作符，可以用来完成3路数值和字符串比较）。如果给定了一个USERSUB，但是该函数未定义，sort会产生一个异常。

为了提高效率，会绕过通常的子例程调用代码，这有以下结果：子例程不能是一个递归子例程（也不能用一个循环控制操作符退出块或例程），要进行比较的两个元素不是通过@\_传入子例程，而是通过临时设置编译sort的包中的全局变量\$a和\$b来完成比较（参见下面的例子）。变量\$a和\$b是真实值的别名，所以不要在子例程中修改这些变量。

比较子例程必须给出一致的结果。如果它返回不一致的结果（例如，有时说\$x[1]小于



`$x[2]`，有时又反过来，说`$x[1]`大于`$x[2]`），结果就不明确（这也是不应修改`$a`和`$b`的另一个原因）。

`USERSUB`可以是一个标量变量名（无下标），在这种情况下，这个值要提供实际使用的子例程的一个符号引用或硬引用（`use strict 'refs' pragma`起作用时，则只允许使用符号名，而不允许使用硬引用）。可以提供一个`BLOCK`作为匿名的内联排序子例程取代`USERSUB`。

要完成一个普通的数值排序，可以写为：

```
sub numerically { $a <=> $b }
@sortedbynumber = sort numerically 53,29,11,32,7;
```

要按降序排序，可以在`sort`之后应用`reverse`，或者可以在排序子例程中把`$a`和`$b`的顺序倒过来：

```
@descending = reverse sort numerically 53,29,11,32,7;

sub reverse_numerically { $b <=> $a }
@descending = sort reverse_numerically 53,29,11,32,7;
```

要按码点顺序对ASCII字符串排序而不考虑大小写，可以在比较之前先用`lc`处理`$a`和`$b`：

```
@unsorted = qw/sparrow Ostrich LARK catbird blueJAY/;
@sorted = sort { lc($a) cmp lc($b) } @unsorted;
```

不同于ASCII，在Unicode中，`lc`和`uc`都不能用于大小写规范化，因为这两个函数无法表述大小写映射的复杂程度。现在有3种大小写形式，而不再是两种，大小写之间不再是一种一对一的映射：也就是说，一些大写字符有多个小写变体，反之亦然。要解决所有这些问题，将来Perl可能会支持一个`fc`函数，之所以这样命名是因为它会生成一个字符串的“casefold”，这正是`/i`模式修饰符使用的形式。估计v5.16左右就会提供`fc`函数，可能通过`use feature "fc"`来提供。如果`fc`函数确实可用，那么在使用`cmp`的排序比较中可以使用这个函数而不是`lc`（假设文本不太复杂，而且你不在意（主要）按数值码点排序）。如果没有这样一个`fc`函数，或者要按字母顺序而不是按码点对文本排序，参见第6章中“Unicode文本比较和排序”一节。

按值对散列排序是`sort`函数的一个常用用法。例如，如果一个`%sales_amount`散列记录部门销售额，在排序例程中完成一个散列查找，根据相应的值对散列键排序：

```
# 从高到低对部门销售额排序
sub bysales { $sales_amount{$b} <=> $sales_amount{$a} }

for $dept (sort bysales keys %sale_amount) {
    say "$dept => $sales_amount{$dept}";
}
```

可以使用`||`或`or`操作符组合多个比较来应用多层排序。这是可行的，因为比较操作符可以很方便地返回0表示相等，从而可以进入下一个比较。在这里散列键首先按相应的销售额排序，然后按键本身排序（万一两个或多个部门有相同的销售额）：

```
sub by_sales_then_dept {
    $sales_amount{$b} <=> $sales_amount{$a}
    ||
    $a cmp $b
}

for $dept (sort by_sales_then_dept keys %sale_amount) {
    say "$dept => $sales_amount{$dept}";
}
```

假设`@recs`是一个散列引用的数组，这里各个散列包含`FIRSTNAME`、`LASTNAME`、`AGE`、`HEIGHT`和`SALARY`等字段。下面的例程会完成排序，先把比较有钱的排在前面，然后看身高，再看年龄，最后按名字的字母顺序排序：

```
sub prospects {
    $b->{SALARY} <=> $a->{SALARY}
    ||
    $b->{HEIGHT} <=> $a->{HEIGHT}
    ||
    $a->{AGE} <=> $b->{AGE}
    ||
    $a->{LASTNAME} cmp $b->{LASTNAME}
    ||
    $a->{FIRSTNAME} cmp $b->{FIRSTNAME}
}

@sorted = sort prospects @recs;
```

从`$a`和`$b`可以得到一些有用的信息，这些信息都可以作为排序例程中比较的基础。例如，如果要根据特定的字段对文本行排序，可以在排序例程中使用`split`来推导得出这些字段。

```
@sorted_lines = sort {
    @a_fields = split /:/, $a;          # 用冒号分隔的字段
    @b_fields = split /:/, $b;

    $a_fields[3] <=> $b_fields[3]        # 按第4个字段进行数值排序，然后
    ||
    $a_fields[0] cmp $b_fields[0]        # 按第1个字段进行字符串排序，然后
    ||
    $b_fields[2] <=> $a_fields[2]        # 按第3个字段进行反向数值排序
    ||
    ...                                  # 等等
} @lines;
```

不过，由于`sort`多次使用不同的`$a`和`$b`值来调用排序例程，前面这个例子会对各行做太多次重新划分，这是没有必要的。

为了避免反复推导（如分解行）来比较字段所带来的开销，可以在排序前对每个值做一次推导，并保存这个推导信息。在这里，会创建一些匿名数组来封装各行以及分解行得到的结果：

```
@temp = map { [$_, split /:/] } @lines;
```

接下来，对数组引用排序：

```
@temp = sort {  
    @a_fields = @a[1..$#a];  
    @b_fields = @b[1..$#b];  
  
    $a_fields[3] <=> $b_fields[3] # 按第4个字段进行数值排序，然后  
    ||  
    $a_fields[0] cmp $b_fields[0] # 按第1个字段进行字符串排序，然后  
    ||  
    $b_fields[2] <=> $a_fields[2] # 按第3个字段进行反向数值排序  
    ||  
    ...                          # 等等  
}  
@temp;
```

既然数组引用已经排序，下面可以从这些匿名数组获取原来的行：

```
@sorted_lines = map { $_->[0] } @temp;
```

把这些汇总起来，这个映射-排序-映射（map-sort-map）技术<sup>注16</sup>可以用一个语句来执行：

```
@sorted_lines = map { $_->[0] }  
    sort {  
        $a->[4] <=> $b->[4] # 当心：索引  
                           # 看起来从1开始  
        ||  
        $a->[1] cmp $b->[1]  
        ||  
        $a->[3] <=> $b->[3]  
        ||  
        ...  
    }  
    map { [$_, split /:/] } @lines;
```

不要用`my`把`$a`和`$b`声明为词法作用域变量。它们是包全局变量（但不受`use strict`对全局变量的限制）。不过，需要确保你的排序例程在同一个包中，否则要用调用者的包名限定`$a`和`$b`。

可以用标准参数传递方法写排序子例程（另外并非巧合，还可以使用XS子例程作为排序子例程），前提是声明排序子例程时提供了原型（`$$`）。如果是这样，实际上可以把`$a`和`$b`声明为词法作用域变量：

---

注16：有时称为施瓦兹变换。



```
sub numerically ($$) {
    my ($a, $b) = @_;
    $a <=> $b;
}
```

另外，将来如果实现了完整的原型，可以写为：

```
sub numerically ($a, $b) { $a <=> $b }
```

这样我们可以算是又回到了起点。

Perl v5.6和更早版本使用了一个快速排序算法来实现排序。这个算法并不是稳定的，其性能可能是二次的（稳定排序是指，相等的元素在排序后会保持原来的输入顺序。对于长度为N的数组，尽管快速排序的运行时平均时间是 $O(N \cdot \log N)$ ，但对于某些输入，排序时间可能达到 $O(N^2)$ ，即二次行为）。在试验性的v5.7版本中，快速排序实现被替换为一种稳定的归并排序算法，其最坏情况行为也是 $O(N \cdot \log N)$ 。不过标准指出，对于某些输入，在某些平台上，原来的快速排序会更快。Perl v5.8有一个`sort pragma`，可以有限地控制排序。它对底层算法的控制很生硬，在将来的Perl版本中可能不会保留，不过在实现中能够以多种独立的可移植方式描述输入或输出，这一点可能会保留。参见第29章的“排序”一节。

## splice



```
splice ARRAY, OFFSET, LENGTH, LIST
splice ARRAY, OFFSET, LENGTH
splice ARRAY, OFFSET
splice ARRAY
```

这个函数从一个数组`ARRAY`中删除`OFFSET`和`LENGTH`指定的元素，并将它们替换为`LIST`的元素（如果提供了`LIST`）。如果`OFFSET`为负，函数从数组末尾倒数，但是如果这会超出数组起始位置，就会产生一个异常。如果`LENGTH`为负，会从`OFFSET`向后删除元素，但保留数组末尾的 $-LENGTH$ 个元素。如果`OFFSET`和`LENGTH`都在列表上下文中，`splice`会返回从数组删除的元素。在标量上下文中，只返回删除的最后一个元素，如果没有删除任何元素则返回`undef`。如果新替换的元素个数不同于被替换的老元素的个数，数组会根据需要扩展或收缩。`splice`后的元素会相应地改变它们的位置。如果省略`LENGTH`，这个函数会删除从`OFFSET`向后的所有元素。如果省略`OFFSET`，则在读取数组时将其清空。如果`OFFSET`和`LENGTH`都省略，则删除所有元素。如果`OFFSET`超过了`ARRAY`的末尾，Perl会发出一个警告，并在`ARRAY`末尾分片。

分片与数组操作的对应如表27-5所示。

表27-5：分片与数组操作的对应

直接方法	等价的分片操作
<code>push(@a, \$x, \$y)</code>	<code>splice(@a, @a, 0, \$x, \$y)</code>
<code>pop(@a)</code>	<code>splice(@a, -1)</code>
<code>shift(@a)</code>	<code>splice(@a, 0, 1)</code>
<code>unshift(@a, \$x, \$y)</code>	<code>splice(@a, 0, 0, \$x, \$y)</code>
<code>\$a[\$x] = \$y</code>	<code>splice(@a, \$x, 1, \$y)</code>
<code>(@a, @a = ())</code>	<code>splice(@a)</code>

`splice`函数对于划分传递给子例程的参数列表也很方便。例如，假设列表长度在列表之前传递：

```
sub list_eq {                                # 比较两个列表值
    my @a = splice(@_, 0, shift);
    my @b = splice(@_, 0, shift);
    return 0 unless @a == @b;                # 长度相同吗?
    while (@a) {
        return 0 if pop(@a) ne pop(@b);
    }
    return 1;
}
if (list_eq($len, @foo[1..$len], scalar(@bar), @bar)) { ... }
```

不过，使用数组引用来完成这个工作更简洁。

从v5.14开始，`splice`可以取一个未祝福的数组的引用，它会自动解引用。`splice`的这个方面还是试验性的。具体的行为在Perl将来的版本中有可能改变。

split



```
split /PATTERN/, EXPR, LIMIT
split /PATTERN/, EXPR
split /PATTERN/
split
```

这个函数扫描`EXPR`给定的一个字符串，检查分隔符，并将这个字符串分解为一个子串列表，在列表上下文中返回得到的列表值，在标量上下文中则返回子串个数<sup>注17</sup>。分隔符由重复的模式匹配确定，会使用`PATTERN`中给定的正则表达式，所以分隔符可以是任意大小，而且不要求每次匹配都是相同的字符串（正常情况下分隔符不会返回，这一节后面会讨论一些例外情况）。如果`PATTERN`与字符串完全不匹配，`split`会把原来的字符串作为子

注17：标量上下文还会导致`split` 将其结果写至`@_`，不过这个用法已经废弃。

串返回。如果匹配一次，则得到两个子串，依此类推。可以为`PATTERN`提供正则表达式修饰符，如`/PATTERN/i`、`/PATTERN/x`等。按模式`/^/`分解时会假设有`//m`修饰符。

如果指定了`LIMIT`而且值为正，这个函数分解的字段数不能超过指定的这个限制（不过如果已经找完所有分隔符，分解得到的字段可能少于这个限制）。如果`LIMIT`为负，会当作好像指定了一个任意大的`LIMIT`。如果省略了`LIMIT`或者`LIMIT`为0，会从结果中去除末尾的`null`字段（`pop`的用户最好记住这一点）。如果省略了`EXPR`，函数会分解`$_`字符串。如果`PATTERN`也省略或者`PATTERN`为直接量空格“ ”，函数会先跳过所有前导空白符，然后按空白符分解`/\s+/`。

如果`PATTERN`为`/^/`，会秘密地作为`/^/m`处理，因为否则没有多大用处。

可以分解任意长的字符串：

```
@chars = split //, $word;
@fields = split /:/, $line;
@words = split " ", $paragraph;
@lines = split /^/, $buffer;
```

使用`split`将字符串分解为一个字形序列是可以的，不过使用模式匹配可以更直接地做到这一点：

```
@graphs = grep { length } split /(\X)/, $word;
@graphs = $word =~ /\X/g;
```

对于能匹配`null`字符串或者比`null`字符串长的某个字符串的模式（例如，由`*`或`?`修饰的单个字符模式），只要它能匹配字符间的`null`字符串，会把`EXPR`的值分解为单个字符；非`null`匹配会像通常那样跳过匹配的分隔符字符（换句话说，一个模式不会在某一点匹配多次，尽管它与0宽度匹配）。例如：

```
print join(":" => split / */, "hi there");
```

会生成输出“`h:i:t:h:e:r:e`”。空格会消失，因为空格会作为分隔符的一部分匹配。来看一个简单的例子，`null`模式`//`只会分解为单个字符，但空格不会消失（在正常的模式匹配中，`//`模式会重复最后一个成功匹配的模式，不过`split`的模式是个例外）。

`LIMIT`参数只分解部分字符串：

```
my ($login, $passwd, $remainder) = split /:/, $_, 3;
```

建议你分解得到类似这样的名字列表，来支持代码的自文档化（完成错误检查时，需要注意如果字段数小于3个，`$remainder`将是未定义的）。赋至一个列表时，如果省略了`LIMIT`，Perl会提供一个`LIMIT`，其值比列表中的变量数多1，以避免不必要的工作。对于上面的分解操作，`LIMIT`默认为4，`$remainder`只接收第3个字段，而不是其余的所有字段。



在时间要求很严格的应用中，最好不要不必要地分解过多字段（强大语言的麻烦之处在于，有时你会过于依赖它的强大而做出一些愚蠢的行为）。

之前我们说过，一般情况下分隔符不会返回，不过如果`PATTERN`包含小括号，那么每对小括号匹配的子串会包含在结果列表中，夹在正常返回的字段中间。

下面是一个简单的例子：

```
split /([-,])/, "1-10,20";
```

这会生成以下列表值：

```
(1, "-", 10, ",", 20)
```

如果有更多小括号，会为每一对小括号返回一个字段，即使有些小括号对不匹配，在这种情况下（即小括号对不匹配时），这些位置上会返回未定义的值。所以，如果有：

```
split /(-)|(/, "1-10,20";
```

会得以下列表值：

```
(1, "-", undef, 10, undef, ",", 20)
```

`/PATTERN/`参数可以替换为一个表达式，来指定运行时可能变化的模式。

作为一个特殊的例子，如果表达式是一个单空格（“ ”），这个函数会按空白符分解，就像无参数的`split`一样。因此，可以用`split(" ")`模拟`awk`的默认行为。与之相反，`split(/ /)`会根据前导空格提供`null`初始字段，有多少前导空格就提供多少个`null`初始字段（除了这种特殊情况，如果你提供了一个字符串而不是正则表达式，它也会解释为一个正则表达式）。可以使用这个属性来删除一个字符串的前导和末尾空白符，另外将中间的连续空白符压缩为一个空格：

```
$string = join(" ", split(" ", $string));
```

下面的例子将一个RFC 822消息首部分解为一个散列，包含`$head{Date}`，`$head{Subject}`等等。这里使用了一个技巧：将一个键/值对列表赋给一个散列，这是因为分隔符与单个字段是交错的。它使用小括号来返回各个分隔符部分，作为返回的列表值的一部分。由于`split`模式肯定会按键/值对返回（因为包含一组小括号），所以散列赋值肯定会接收一个包含键/值对的列表，其中各个键就是首部字段的名称（遗憾的是，对于有相同键字段的多行文本，如`Received-By`行，使用这个技术会丢失信息。嗯，没办法）。

```
$header =~ s/\n\s+/ /g;      # Merge continuation lines.
%head = ("FRONTSTUFF", split /^(\S*?):\s*/m, $header);
```

下面的例子处理一个Unix *passwd(5)*文件中的记录。可以去掉`chomp`，如果没有`chomp`，`$shell`末尾会有一个换行符。

```
open(PASSWD, "/etc/passwd");
while (<PASSWD>) {
    chomp;          # 删除末尾换行符
    ($login, $passwd, $uid, $gid, $gcos, $home, $shell) =
        split /:/;
    ...
}
```

可以如下处理各个输入文件中每一行中的各个单词，来创建一个单词频度散列。

```
while (<>) {
    for my $word (split) {
        $count{$word}++;
    }
}
```

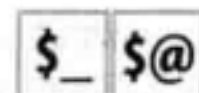
`split`的逆操作是`join`，不过`join`只能用相同的分隔符连接所有字段。要分解包含固定位置字段的字符串，需要使用`unpack`。

## sprintf

`sprintf` *FORMAT*, *LIST*

这个函数返回一个格式化的字符串，它按C的库函数`sprintf`常用的`printf`约定完成格式化。参见你的系统上的`sprintf(3)`或`printf(3)`，了解一般原则的有关解释。*FORMAT*字符串包含文本，其中有嵌入的字段指示符，这些指示符将替换为*LIST*的元素，对应各个字符分别有一个指示符。关于字段的解释，参见第26章“字符串格式”一节。

## sqrt



`sqrt` *EXPR*  
`sqrt`

这个函数返回*EXPR*的平方根。要得到其他根式，如立方根，可以使用`**`操作符来得到一个数的分数次幂。不要试图对负数使用这些方法，因为这会带来一个更复杂的问题（并产生一个异常）。不过有一个标准模块专门负责这个任务；

```
use Math::Complex;
print sqrt(-2);      # 打印1.4142135623731i
```

## srand

`srand` *EXPR*  
`srand`

这个函数为rand操作符设置随机数种子。如果忽略EXPR，它会使用内核提供的一个半随机值（如果支持/dev/urandom设备），或者根据当前时间和进程ID（以及其他方面）来提供随机数种子。不论哪一种情况，从v5.14开始，它都会返回这个种子。通常根本没有必要调用srand，因为即使没有显式调用，第一次使用rand操作符时也会隐式地调用srand。不过，v5.004 (1997)之前的Perl版本中不是这样，所以如果你的脚本需要在较老的Perl版本下运行，则应当调用srand。

对于一些频繁调用的程序（如CGI脚本），如果只使用时间 ^ \$\$作为种子，那么三分之一的情况下都可能会深受数学特性 $a^b == (a+1)^{(b+1)}$ 之苦。所以不要这么做，而应当使用：

```
srand( time() ^ ($$ + ($$ << 15)) );
```

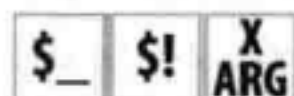
要完成加密，需要比默认种子更大的随机性。在一些系统上，/dev/random设备就很合适。否则，可以取一个或多个会快速改变操作系统状态的程序的输出，压缩这些输出并计算校验和，这也是一个常用的方法。例如：

```
srand (time ^ $$ ^ unpack "%32L*", `ps wwaxl | gzip`);
```

如果你很关注这个方面，可以参见CPAN中的Math::TrulyRandom模块。

不要在程序中多次调用srand，除非你很清楚自己在做什么，以及为什么这么做。这个函数的重点是为rand函数提供“种子”，使得rand能够在每次运行程序时生成一个不同的序列。要在程序最前面调用srand，否则无法从rand得到随机数！

## stat



```
stat FILEHANDLE  
stat DIRHANDLE  
stat EXPR  
stat
```

在标量上下文中，这个函数返回一个布尔值，指示调用是否成功。在列表上下文中，它返回一个包含13个元素的列表来提供一个文件的统计信息，这可能是通过FILEHANDLE或DIRHANDLE打开的文件，或者是由EXPR指定的文件。通常如下使用：

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,$size,  
 $atime,$mtime,$ctime,$blksize,$blocks)  
    = stat $filename;
```

并不是所有字段在所有类型的文件系统上都得到支持；不支持的字段会返回0。

表27-6列出了这些字段的含义。



表27-6: stat返回的字段

索引	字段	含义
0	\$dev	文件系统设备号
1	\$ino	Inode号
2	\$mode	文件模式（类型和权限）
3	\$nlink	文件（硬）链接数
4	\$uid	文件所有者的数值用户ID
5	\$gid	文件所属组的数值组ID
6	\$rdev	设备标识符（只用于特殊文件）
7	\$size	文件总大小（字节数）
8	\$atime	自纪元以来的最后访问时间（秒数）
9	\$mtime	自纪元以来的最后修改时间（秒数）
10	\$ctime	自纪元以来的Inode变更时间（非创建时间），秒数
11	\$blksize	文件系统I/O的首选块大小
12	\$blocks	分配的实际块数

\$dev和\$ino合在一起共同唯一地标识一个系统上的文件。可能只有BSD系列的文件系统上定义了\$blksize和\$blocks。\$blocks字段（如果定义）按512字节的块来报告块数。如果文件中包含未分配的块或者“洞”（这些不会统计在\$blocks中），\$blocks\*512的值与\$size会有很大不同。

如果向stat传入一个特殊文件句柄\_，不会完成具体的stat(2)，而会返回最后一个stat、lstat或基于stat的文件测试操作符（如-r、-w和-x）得到的stat结构的当前内容。

由于模式包含文件类型及其权限，如果你想看到实际的权限，应当屏蔽文件类型部分，使用“%o”来调用printf或sprintf：

```
$mode = (stat($filename))[2];
printf "Permissions are %04o\n", $mode & 07777;
```

File::stat模块提供了一个很方便的机制，可以按名访问：

```
use File::stat;
$sb = stat($filename);
printf "File is %s, size is %s, perm %04o, mtime %s\n",
    $filename, $sb->size, $sb->mode & 07777,
    scalar localtime $sb->mtime;
```

还可以从Fcntl模块导入各个模式位的符号定义。

```
use Fcntl ':mode';
```

```

$mode = (stat($filename))[2];

$user_rwx      = ($mode & S_IRWXU) >> 6;
$group_read    = ($mode & S_IRGRP) >> 3;
$other_execute = $mode & S_IXOTH;

printf "Permissions are %04o\n", S_IMODE($mode), "\n";

$is_setuid = $mode & S_ISUID;
$is_directory = S_ISDIR($mode);

```

最后两个也可以使用-u和-d操作符来写。更多详细信息参见`stat(2)`。

提示：如果只需要文件的大小，可以考虑使用-s文件测试操作符，它会直接返回文件大小（字节数）。另外还有一些文件测试操作符可以返回文件的年龄（天数）。

## state

```

state EXPR
state TYPE EXPR
state EXPR : ATTRS
state TYPE EXPR : ATTRS

```

`state`声明符可以引入一个词法作用域变量，就像`my`一样。不过，在多次调用同一个例程之间，状态变量的内容会保留。这些变量只能在第一次进入作用域时初始化一次，然后不会再重新初始化，这一点不同于词法作用域变量，后者每次进入其外围作用域时都会重新初始化。

克隆一个闭包时，会认为是一个新的子例程，所以在第一次调用时，所有状态变量都会在这个新的克隆中初始化。状态变量不是C程序员所认为的那种静态变量，除非例程本身是静态的。

状态变量只在`use feature "state" pragma`起作用时才启用。参见第29章“feature”一节。目前只有标量状态变量的初始化得到充分支持，不过完全可以使用数组或散列的标量引用。

## study

\$ \_

```

study SCALAR
study

```

这个函数花额外的时间来研究`SCALAR`，因为预期到下一次修改这个字符串之前可能还会对它完成多次模式匹配。也许这会节省时间，也可能不会节省，这取决于你搜索的模式性质和数量，以及字符串中要搜索的字符频率分布，你可能想做一个比较，即在执行这个函数和不执行这个函数的不同情况下比较运行时间，来看哪一个运行得更快。如果循环中要扫描多个短常量字符串（包括更复杂的模式中的常量部分），这些循环最能从`study`受

益。如果所有的模式匹配都是位于最前面的常量字符串，`study`就一点帮助也没有，因为没有做任何扫描。一次只能有一个活动的`study`。如果在研究另一个不同的标量，那么前一个就是“未研究的”。

`study`的工作原理如下：建立一个链表，其中包含要搜索的字符串中的各个字符，这样一来我们可以知道各个字符的信息，举例来说，可以知道所有“k”字符都在哪里。根据静态频度表从各个搜索字符串中选择出现最少的字符，静态频度表由某些C程序和英语文本来构建。只对这个最少字符出现的位置进行检查。

例如，下面这个循环会在包含某个模式的所有行前面插入索引生成的一些记录：

```
while (<>) {
    study;
    print ".IX foo\n"      if /\bfoo\b/;
    print ".IX bar\n"      if /\bbar\b/;
    print ".IX blurfl\n"   if /\bglarch\b/;
    ...
    print;
}
```

搜索`/\bfoo\b/`时，只会查看`$_`中包含“f”的位置，因为“f”比“o”少。除了一些特殊情况，一般来讲这会有很大好处。唯一的问题是，这样节省的时间是否比开始时构建这个链表花费的时间更多。

如果要查找的字符串直到运行时才能知道，可以把整个循环建立一个字符串，并用`eval`执行这个字符串，来避免每次都得重新编译所有模式。同时将`$/`设置为输入整个文件作为一个记录，这样可能运行速度很快，通常比一些专用的程序（如`fgrep(1)`）还要快。下面会扫描一个文件列表（`@files`）来查找一个单词列表（`@words`），并打印包含匹配（不区分大小写）的那些文件的名字：

```
$search = "while (<>) { study;";
for my $word (@words) {
    $search .= "++\${seen}{\$ARGV} if /\b$word\b/i;\n";
}
$search .= "}";
@ARGV = @files;
undef $/;          # 吞入整个文件
eval $search;       # 运行程序
die $@ if $@;       # 如果eval失败
$/ = "\n";          # 恢复正常的输入终止符
for my $file (sort keys(%seen)) {
    say $file;
}
```

既然有`qr//`操作符，上面看到的复杂的运行时`eval`就不太必要了。下面的代码可以做同样的事情：

```
@pats = ();
```



```

for my $word (@words) {
    push @pats, qr/\b${word}\b/i;
}
@ARGV = @files;
undef $/;                # 吞入整个文件
while (<>) {
    for $pat (@pats) {
        $seen{$ARGV}++ if /$pat/;
    }
}
$/ = "\n";                # 恢复正常的输入终止符
for my $file (sort keys(%seen)) {
    say $file;
}

```

## sub

命名声明：

```

sub NAME PROTO ATTRS
sub NAME ATTRS
sub NAME PROTO
sub NAME

```

命名定义：

```

sub NAME PROTO ATTRS BLOCK
sub NAME ATTRS BLOCK
sub NAME PROTO BLOCK
sub NAME BLOCK

```

未命名定义：

```

sub PROTO ATTRS BLOCK
sub ATTRS BLOCK
sub PROTO BLOCK
sub BLOCK

```

子例程声明和定义的语法看起来很复杂，不过实际上相当简单。所有这些都基于下面这个语法：

```
sub NAME PROTO ATTRS BLOCK
```

所有这4个字段都是可选的。唯一的限制是如果在某个字段就必须按这个顺序出现，而且至少要有一个NAME或BLOCK。目前，我们会忽略PROTO和ATTRS；它们只是基本语法的修饰符。NAME和BLOCK是声明和定义的重要部分：

- 如果只有一个NAME而没有BLOCK，这是指定名字的一个预声明（不过，如果你想调用这个子例程，以后还必须用NAME和一个BLOCK提供一个定义）。命名声明很有用。因为解析器如果知道这是一个用户自定义的子例程，它会以特殊的方式处理这个名

字。 这样的子例程可以作为一个函数或操作符来调用，就像内置函数一样。有时这称为前置（forward）声明。

- 如果同时有一个`NAME`和一个`BLOCK`，这是一个标准的命名子例程定义（如果之前没有声明这个名字，那么这还是一个标准的声明）。命名定义很有用，因为`BLOCK`会为此声明关联一个具体含义（子例程的体）。这就是我们所说的“定义了一个子例程”而不只是“声明了一个子例程”。不过，定义与声明也有类似的地方，周围的代码看不到它，而且不会返回可以用来引用子例程的内联值。
- 如果只有一个`BLOCK`而没有`NAME`，这就是一个匿名定义。也就是说，是一个匿名子例程。由于它没有名字，所以这根本不是一个声明，而是一个真正的操作符，会返回运行时匿名子例程体的一个引用。这对于将代码处理为数据极其有用。它允许你传入奇怪的代码块作为回调，如果`sub`定义操作符指示它自身以外的某些词法作用域变量，甚至可以用作为闭包。这说明同一个`sub`操作符的不同调用会做必要的维护工作，以保证每一个这种词法作用域变量的正确“版本”在闭包生命期中都可见，即使这个词法作用域变量原来的作用域已经撤销。

在这3种情况下，`PROTO`和`ATTRS`中任意一个（或二者）可以出现在`NAME`之后，但必须在`BLOCK`前面。原型`PROTO`是放在小括号里的一个字符列表，告诉解析器如何处理函数的参数。属性`ATTRS`由一个冒号引入，向解析器提供有关这个函数的额外信息。下面是一个典型的定义，其中包括所有这4个字段：

```
sub numstrcmp ($$) : locked {  
    my ($a, $b) = @_;  
    return $a <=> $b || $a cmp $b;  
}
```

关于属性列表及其管理的详细内容，参见第29章“attributes”一节。另外参见第7章以及第8章中“匿名子例程生成器”一节。

## substr



```
substr EXPR, OFFSET, LENGTH, REPLACEMENT  
substr EXPR, OFFSET, LENGTH  
substr EXPR, OFFSET
```

这个函数从`EXPR`给定的字符串抽取一个子串并返回这个子串。子串从这个字符串前面的`OFFSET`字符开始抽取。如果`OFFSET`为负，子串则从字符串末尾的相应位置开始抽取。如果省略`LENGTH`，会返回直到字符串末尾的所有字符。如果`LENGTH`为负，这个长度则是要从字符串末尾保留的字符个数。否则，`LENGTH`指示要抽取的子串长度，这通常是你想做的。

注意，我们谈到字符时，是指码点而不是字节或字形。对于字节，首先会编码为UTF-8，然后再尝试取子串。对于字形，要使用CPAN `Unicode::GCString`模块的`substr`方法。

可以使用`substr`作为一个左值（可以为它赋值），在这种情况下，*EXPR*必须是一个合法的左值。如果所赋的字符串比你的子串长度短，这个字符串会收缩，如果所赋的字符串比你的子串长度长，这个字符串会相应扩展。为了保证字符串长度相同，需要使用`sprintf`或`x`操作符填充或删除值。

如果想为超过字符串末尾的一个未分配的区域赋值，`substr`会产生一个异常。

要把字符串“Larry”追加到`$_`当前值的前面，可以使用：

```
substr($var, 0, 0) = "Larry";
```

要用“Moe”替换`$_`的第一个字符，可以使用：

```
substr($var, 0, 1) = "Moe";
```

最后，要用“Curly”替换`$var`的最后一个字符，可以使用：

```
substr($var, -1) = "Curly";
```

要把`substr`作为一个左值使用，还有一个候选方法，可以指定`REPLACEMENT`字符串作为第4个参数。这允许你替换`EXPR`的某些部分，并返回被替换的部分，所有这些都在一个操作中完成，就像使用`splice`一样。下面的例子也是用“Curly”替换`$var`的最后一个字符，并把被替换的字符放在`$oldstr`中：

```
$oldstr = substr($var, -1, 1, "Curly");
```

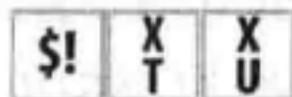
并非只能对赋值使用左值`substr`。下面的代码会用点号替换所有空格，不过只处理字符串中最后10个字符：

```
substr($var, -10) =~ s/ /. /g;
```

需要说明，我们一直在说字符，与本书中其他地方一样，这里我们指的是码点，即程序员眼里的字符，而不是字形（用户眼里的字符），字形可能（而且通常）会包含多个码点。CPAN `Unicode::GCString`模块为`substr`、`index`、`pos`和很多其他函数提供了替代函数，从而可以按逻辑字形处理字符串，而不是基于码点进行处理。

如果你想用`substr`而不愿意使用正则表达式，因为你认为`substr`肯定更快一些，可能会惊讶地发现并非如此。通常，正则表达式会比`substr`更快，甚至对定宽的字段也是如此。

## symlink



```
symlink OLDNAME, NEWNAME
```

这个函数创建一个新的文件名，以符号链接方式链接到原来的文件名。这个函数成功时返回`true`，否则返回`false`。在不支持符号链接的系统上，会在运行时产生一个异常。要检查系统是否支持符号链接，可以使用`eval`捕获可能的错误：



```
$can_symlink = eval { symlink("", ""); 1 };
```

或者使用Config模块。如果提供了一个相对符号链接，要当心，因为它会相对于符号链接本身的位置来解释，而不是相对于你的当前工作目录。

参见这一章前面的link和readlink。

## syscall



syscall LIST

这个函数调用一个列表中第一个元素指定的系统调用（这是指系统的调用，而不是一个shell命令），并把其余的元素作为参数传递给这个系统调用（现在可以通过类似POSIX的模块得到很多这样的调用）。如果未实现syscall(2)，这个函数会产生一个异常。

参数的解释如下：如果给定参数是数值，这个参数会作为一个C整数传递。如果不是，则传递这个字符串值的一个指针。你要负责确保这个字符串足够长，可以接收可能写入的任何结果；否则，你会看到一个内核转储。不能使用一个字符串直接量（或者其他只读字符串）作为syscall的参数，因为Perl必须假设所有字符串指针都可以写入。如果你的整数参数不是直接量，而且未在一个数值上下文中解释，可能需要为它们加0，强制地使它们看上去像是数字。

syscall返回所调用的系统调用返回的值。按照C编码约定，如果那个系统调用失败，syscall返回-1，并设置\$(错误号)。有些系统调用成功时会返回-1。处理这种调用的适当方法是在调用前赋值\$!=0，然后当syscall返回-1时检查\$!的值。

并不是所有系统调用都能以这种方式访问。例如，Perl支持为系统调用传递最多14个参数，在实际中这往往足够了。不过，对于返回多个值的syscall还存在一个问题。考虑syscall(&SYS\_pipe)：它会返回所创建的管道读取端的文件号。但没有办法获取另一端的文件号。对于这个特定的问题可以用pipe来避免。要解决这个一般问题，可以写一个XSUB（外部子例程模块，这是C的一种说法）直接访问这些系统调用。然后把你的新模块放在CPAN上，让它流行起来。

下面的子例程将当前时间作为一个浮点数返回，而不像time只返回整数秒数（只有支持gettimeofday(2)系统调用的机器才能使用这个子例程）。

```
sub finetime() {
    package main;      # 用于下一个require
    require "syscall.ph";
    # 预分配缓冲区大小为两个32位...
    my $tv = pack("LL", ());
    syscall(&SYS_gettimeofday, $tv, undef) >= 0
        || die "gettimeofday: $!";
    my($seconds, $microseconds) = unpack("LL", $tv);
```

```

        return $seconds + ($microseconds / 1_000_000);
    }

```

假设Perl不支持`setgroups(2)`系统调用<sup>注18</sup>，但你的内核支持。可以这样来实现：

```

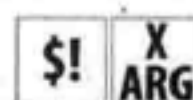
require "syscall.ph";
syscall(&SYS_setgroups, scalar @newgids, pack("i*", @newgids))
    || die "setgroups: $!";

```

可能必须按照Perl安装说明中所指示的，要运行`h2ph`来检查`syscall.ph`是否存在。有些系统可能需要一个`pack`模板“II”。更烦人的是，`syscall`假设C类型`int`、`long`和`char*`的大小是相同的。不要认为`syscall`体现了可移植性。

关于细粒度计时问题，还有一种更严格的方法，参见CPAN的`Time::HiRes`模块。

## sysopen



```

sysopen FILEHANDLE, FILENAME, MODE, MASK
sysopen FILEHANDLE, FILENAME, MODE

```

`sysopen`函数打开文件名由`FILENAME`指定的一个文件，并将它与`FILEHANDLE`关联。如果`FILEHANDLE`是一个表达式，它的值会用作这个文件句柄的名字或引用。如果`FILEHANDLE`是一个变量，其值未定义，则会为你创建一个值。如果调用成功，返回值为`true`，否则返回`false`。

这个函数是一个直接接口，包括操作系统的`open(2)`系统调用以及后面的一个`fdopen(3)`库调用。因此，这里你要暂时假装自己是一个C程序员。`MODE`参数的可取值和标志位可以通过`Fcntl`模块得到。因为不同的系统支持不同的标志，所以不要指望所有这些标志在你的系统上都可用。参考你的`open(2)`手册页或相应的本地文档来了解详细信息。不过，带有一个标准C库的系统上应该都支持表27-7所列的标志。

表27-7: `sysopen`的标志

标志	含义
<code>O_RDONLY</code>	只读
<code>O_WRONLY</code>	只写
<code>O_RDWR</code>	读写
<code>O_CREAT</code>	如果文件不存在则创建文件
<code>O_EXCL</code>	如果文件已经存在则失败
<code>O_APPEND</code>	追加到文件

注18：不过通过`$()`是支持的。

表27-7：sysopen的标志

标志	含义
O_TRUNC	清除文件
O_NONBLOCK	非阻塞访问

不过还有很多其他选项。表27-8列出了一些不太常用的标志。

表27-8：sysopen不太常用的标志

标志	含义
O_NDELAY	原来O_NONBLOCK的同义词
O_SYNC	写阻塞，直到数据物理写至底层硬件。还可以参见O_ASYNC，O_DSYNC和O_RSYNC
O_EXLOCK	带LOCK_EX的flock（只是建议性锁）
O_SHLOCK	带LOCK_SH的flock（只是建议性锁）
O_DIRECTORY	如果文件不是一个目录则失败
O_NOFOLLOW	如果最后一个路径分量是一个符号链接则失败
O_BINARY	为Microsoft系统的句柄设置binmode。有时还会存在一个O_TEXT来得到相反的行为
O_LARGEFILE	有些系统需要这个标志来处理超过2GB的文件
O_NOCTTY	打开一个终端文件不会让这个终端成为进程的控制终端（如果还没有一个控制终端）。一般情况下这个标志已经不再需要

O\_EXCL标志并不是用于锁定：在这里，排他性表示如果文件已经存在，则sysopen失败。

如果名为FILENAME的文件不存在，而且MODE包含O\_CREAT标志，sysopen会创建这个文件，初始权限由MASK参数确定（或者如果省略了MASK参数，则为0666），权限由进程的当前umask修改。这种默认设置是合理的：参见umask来了解相关解释。

用open和sysopen打开的文件句柄可以交替使用。如果你打开文件时使用的是sysopen，并不要求之后必须使用sysread及类似操作，如果你用open打开文件，然后也完全可以使用sysread及类似操作。它们分别可以做一些对方做不到的事情。常规的open可以打开管道、派生子进程、设置层、复制文件句柄，以及将一个文件描述符编号转换为一个文件句柄。它还会忽略文件名中的前导和末尾空白符，并把“-”当作一个特殊文件名。不过在打开实际的文件方面，open能做的，sysopen都可以做到。

下面的例子显示了两个函数的等价调用。为简洁起见，我们省略了or die \$!检查，不过你的程序中一定要检查返回值。这里我们有所限制，只使用了几几乎所有操作系统上都可用



的那些标志。实际上就是控制传入`MODE`参数的值，要用位操作符“`|`”将这些值OR（逻辑或）在一起。

- 打开一个文件用于读：

```
open(FH, "<", $path);  
sysopen(FH, $path, O_RDONLY);
```

- 打开一个文件用于写，如果必要还需要创建一个新文件，或者清除一个原有的文件：

```
open(FH, ">", $path);  
sysopen(FH, $path, O_WRONLY | O_TRUNC | O_CREAT);
```

- 打开一个文件用于追加，如果必要则要创建一个新文件：

```
open(FH, ">>", $path);  
sysopen(FH, $path, O_WRONLY | O_APPEND | O_CREAT);
```

- 打开一个文件用于更新，要求文件必须已经存在：

```
open(FH, "+<", $path);  
sysopen(FH, $path, O_RDWR);
```

以下工作可以使用`sysopen`完成，但用常规的`open`无法做到：

- 打开并创建一个文件用于写，要求之前不存在这个文件：

```
sysopen(FH, $path, O_WRONLY | O_EXCL | O_CREAT);
```

- 打开一个文件用于追加，要求文件必须已经存在：

```
sysopen(FH, $path, O_WRONLY | O_APPEND);
```

- 打开一个文件用于更新，如果必要需要创建一个新文件：

```
sysopen(FH, $path, O_RDWR | O_CREAT);
```

- 打开一个文件用于更新，要求之前不存在这个文件：

```
sysopen(FH, $path, O_RDWR | O_EXCL | O_CREAT);
```

- 以非阻塞模式打开一个只写文件，但是如果文件不存在并不创建新文件：

```
sysopen(FH, $path, O_WRONLY | O_NONBLOCK);
```

`IO::File`模块为打开文件提供了一组面向对象同义词（以及一点新功能）。你可以在用`open`、`sysopen`、`pipe`、`socket`或`accept`创建的任何句柄上调用适当的`IO::File`或`IO::Handle`方法，尽管并没有用这个模块来初始化那些句柄。实际上，Perl现在会根据需要隐式地加载这些模块来确保你可以使用这些方法。

## sysread

\$!	\$@	T	X ARG	X RO
-----	-----	---	-------	------

```
sysread FILEHANDLE, SCALAR, LENGTH, OFFSET  
sysread FILEHANDLE, SCALAR, LENGTH
```

这个函数尝试从指定的`FILEHANDLE`使用一个底层系统调用`read(2)`将`LENGTH`个数据字符读入变量`SCALAR`。这个函数返回读取的字符个数，或者如果达到EOF则返回0<sup>注19</sup>。如果有错误，`sysread`函数返回`undef`。`SCALAR`会随实际读取的长度扩展或收缩。如果指定了`OFFSET`，这是指从字符串的哪个位置开始放入字符，因此可以向一个字符串（用作缓冲区）中间读入数据。要看一个使用`OFFSET`的例子，参见`syswrite`。如果`LENGTH`为负或者如果`OFFSET`指向字符串以外，则会产生一个异常。

如果文件句柄没有编码层，读入的字符不会大于255，所以它们实际上是字节。

要做好准备处理标准I/O通常会为你处理的那些问题（比如中断的系统调用）。因为它绕过了标准I/O，不要在同一个文件句柄上混合使用`sysread`和其他读操作（`print`、`printf`、`write`、`seek`、`tell`或`eof`），除非你想自找麻烦。另外，从一个包含UTF-8、UTF-16或另外某种多字节编码的文件中读取字符时，缓冲区边界可能会落在一个字符的中间。因此最好设置编码，另外要读取字符而不是字节。

需要说明，如果文件句柄已经标志为`:utf8`，会读取Unicode字符而不是字节（`LENGTH`、`OFFSET`和`sysread`的返回值都按Unicode字符考虑）。`:encoding(...)`层会隐式地引入`:utf8`层。

## sysseek

\$!	X ARG
-----	-------

```
sysseek FILEHANDLE, POSITION, WHENCE
```

这个函数使用系统调用`lseek(2)`设置`FILEHANDLE`的系统位置。它绕过了标准I/O，所以将这个函数与其他读操作（除了`sysread`以外，包括`print`、`write`、`seek`、`tell`或`eof`）混合使用可能（而且很可能会）导致混乱。`FILEHANDLE`可以是一个表达式，这个表达式的值给出了文件句柄的名字。`WHENCE`的值如果为0，则设置新位置为文件中的`POSITION`字节，如果为1则将新位置设置为当前位置加上`POSITION`，如果`WHENCE`的值为2，则将新位置设置为EOF加上`POSITION`字节（通常为负）。`WHENCE`可以使用标准IO::Seekable和POSIX模块或者Fcntl模块的常量`SEEK_SET`、`SEEK_CUR`和`SEEK_END`，这更可移植，更方便。

这个函数按字节数返回新位置，如果失败则返回`undef`。位置为0会返回为特殊字符串“`o but true`”，这可以作为数值使用而不会产生警告，另外不要求必须使用`// die`，可以用

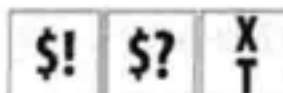
---

注19：不存在一个`sys eof`函数，这是有道理的，因为`eof`不能很好地处理设备文件（如终端）。应当使用`sysread`并检查返回值0来确定是否结束。

更常用的`die`。

警告：不论文件句柄上是否有编码层，`POSITION`都按字节统计而不是按字符统计。不过，Perl中所有读文件的函数确实都会通过某个编码层，因此可能读到一个不完整的部分“字符”，并形成一个不合法的Perl字符串。在有多字节编码层的文件句柄上，要避免混合使用`sysseek`或`seek`调用和I/O函数。

## system



```
system PATHNAME LIST
system LIST
```

这个函数在系统上为你执行程序，并返回该程序的退出状态，而不是其输出。要捕获一个命令的输出，可以使用反引号或`qx//`。`system`函数的工作与`exec`非常类似，不过`system`首先会执行一个`fork`，然后在`exec`之后等待所执行的程序结束。也就是说，它会为你运行这个程序，并在程序结束时返回，而`exec`会用新程序替代你当前运行的程序，所以如果替换成功，它不会返回。

取决于参数个数的不同，参数处理会有所变化，与`exec`的有关介绍一样，这包括确定是否调用shell，以及是否指定一个单独的`PATHNAME`为程序给出名字。由于`system`和反引号会阻塞`SIGINT`和`SIGQUIT`，向正在运行的程序发送这些信号（如从Control-C）不会中断你的主程序。但是你运行的另一个程序确实会得到这个信号。检查`system`的返回值来查看你运行的程序是否正常退出。

```
@args = ("command", "arg1", "arg2");
system(@args) == 0
    || die "system @args failed: $?"
```

返回值是程序通过`wait(2)`系统调用返回时的退出状态。基于传统语义，要得到真正的退出值，需要除以256或右移8位。这是因为低字节包含其他的信息（实际上有两个信息）。低7位指示关闭进程的信号编号（如果有），第8位指示这个进程是否转储内核。通过查看`$?`（`$CHILD_ERROR`），可以检查所有失败的可能性，包括信号和内核转存：

```
$exit_value = $? >> 8;
$signal_num = $? & 127;      #或0x7f,或0177,或0b0111_1111
$dumped_core = $? & 128;     #或0x80,或0200,或0b1000_0000
```

如果程序必须通过系统shell运行<sup>注20</sup>，因为你只有一个参数，而且这个参数中包含shell元字符，通常返回码要受这个shell的额外问题和功能的影响。换句话说，如果是这样，你可能无法恢复之前介绍的详细信息。

---

注20：根据定义，系统shell是`/bin/sh`，或者是你的平台上的任何有意义的shell，但不是用户当时恰好正在使用的shell。



## syswrite

\$!	\$@	X ARG	X WIDE
-----	-----	----------	-----------

```
syswrite FILEHANDLE, SCALAR, LENGTH, OFFSET
syswrite FILEHANDLE, SCALAR, LENGTH
syswrite FILEHANDLE, SCALAR
```

这个函数尝试使用`write(2)`系统调用从变量`SCALAR`向指定的`FILEHANDLE`写入`LENGTH` 个字节的数据。这个函数返回所写的字节数，如果有错误，则返回`undef`。如果指定了`OFFSET`，这是指从字符串的哪个位置开始写（例如，你可以认为你在使用字符串作为一修缮区，或者需要从一个部分写恢复）。`OFFSET`为负指定要从字符串末尾倒数指定的字节数开始写。如果`SCALAR`为空，则只允许`OFFSET`为0。如果`LENGTH`为负或者如果`OFFSET`指向字符串以外，会产生一个异常。

要从文件句柄FROM向文件句柄TO复制数据，可以使用类似下面的代码：

```
use Errno qw/EINTR/;
$blksize = (stat FROM)[11] || 16384; # 首选块大小?
while ($len = sysread FROM, $buf, $blksize) {
    if (!defined $len) {
        next if $! == EINTR;
        die "System read error: $!";
    }
    $offset = 0;
    while ($len) { # 处理部分写
        $written = syswrite TO, $buf, $len, $offset;
        die "System write error: $!" unless defined $written;
        $offset += $written;
        $len -= $written;
    }
}
```

必须做好准备处理标准I/O通常会为你处理的那些问题，如部分写。因为`syswrite`会绕过C标准I/O库，所以不要将`syswrite`调用与读操作（除`sysread`以外）、写操作（如`print`、`printf`或`write`）或者其他`stdio`函数（如`seek`、`tell`或`eof`）混合使用，除非你想自找麻烦<sup>注21</sup>。

如果文件句柄标志为`:utf8`，会写入UTF-8编码的Unicode字符而不是字节，另外`LENGTH`、`OFFSET`和`syswrite`的返回值都按（UTF-8编码的Unicode）字符考虑。`:encoding(...)`层会隐式地引入`:utf8`层。

## tell

X ARG
----------

```
tell FILEHANDLE
tell
```

---

注21：而且不怕痛苦。

这个函数返回`FILEHANDLE`的当前文件位置（基于0的字节数）。一般地，将来可以把这个值输入到`seek`函数来返回当前位置。`FILEHANDLE`可以是一个表达式，给出实际文件句柄的名字，或者是一个文件句柄对象的引用。如果省略`FILEHANDLE`，这个函数会返回最后读取的文件的位置。文件位置只对常规文件有意义。设备、管道和套接字都没有文件位置。

注意这里强调按字节数指定位置：即使文件句柄已经设置为处理字符（例如，通过使用`:encoding(utf8)` 层），`tell`仍会返回字节偏移量，而不是字符偏移量（因为这会使`seek`和`tell`相当慢）。

并没有一个`sys tell`函数。要实现这一点可以使用`sys seek(FH, 0, 1)`。可以查看`seek`下的例子来了解如何使用`tell`。

不要在一个由`sys read`、`sys write`或`sys seek`处理的文件句柄上使用`tell`（或任何其他缓冲I/O操作）。这些函数会忽略缓冲，而`tell`不会。

## telldir



`telldir DIRHANDLE`

这个函数返回`DIRHANDLE`上`readdir`例程的当前位置。这个值可以提供给`seekdir`，来访问目录中的一个特定位置。这个函数与相应的系统库例程在可能的目录压缩方面有同样的问题。实现`readdir`的系统不一定也实现了这个函数，即使实现了这个函数，你也不能对返回值做任何计算。这只是一个不透明的值，只对`seekdir`有意义。

## tie



`tie VARIABLE, CLASSNAME, LIST`

这个函数将一个变量绑定到一个包类，这个类为这个变量提供了实现。`VARIABLE`是要绑定的变量（标量、数组或散列）或类型团（表示一个文件句柄）。`CLASSNAME`是实现了一个适当类型的对象的类名。

所有额外的参数会传递给这个类的适当的构造方法，可能是`TIESCALAR`、`TIEARRAY`、`TIEHASH`或`TIEHANDLE`（如果没有找到适当的方法，则会产生一个异常）。一般地，这些参数可能与传递给C的`dbm_open(3)`函数的参数类似，不过其含义取决于具体的包。构造函数返回的对象再由`tie`函数返回，如果你想访问`CLASSNAME`中的另一个方法，这就很有用（还可以通过`tied`函数访问这个对象）。所以，如果一个类要把一个散列绑定到一个I SAM实现，可能会提供一个额外的方法顺序地遍历一组键（I SAM中的“S”就表示“顺序地”（sequentially）），因为典型的DBM实现不会这么做。

对于类似DBM文件等很大的对象，在这些对象上使用`keys`和`values`等函数时可能会返回

超大的列表值。你可能更愿意使用each函数来迭代处理这种列表。例如：

```
use NDBM_File;
tie(%ALIASES, "NDBM_File", "/etc/aliases", 1, 0)
|| die "Can't open aliases: $!";
while (($key,$val) = each %ALIASES) {
    say "$key = $val";
}
untie %ALIASES;
```

实现散列的类应当提供以下方法：

```
TIEHASH CLASS, LIST
FETCH SELF, KEY
STORE SELF, KEY, VALUE
DELETE SELF, KEY
CLEAR SELF
EXISTS SELF, KEY
FIRSTKEY SELF
NEXTKEY SELF, LASTKEY
SCALAR SELF
DESTROY SELF
UNTIE SELF
```

实现数组的类应当提供以下方法：

```
TIEARRAY CLASS, LIST
FETCH SELF, KEY
STORE SELF, KEY, VALUE
FETCHSIZE SELF
STORESIZE SELF, COUNT
CLEAR SELF
PUSH SELF, LIST
POP SELF
SHIFT SELF
UNSHIFT SELF, LIST
SPLICE SELF, OFFSET, LENGTH, LIST
EXTEND SELF, COUNT
DESTROY SELF
UNTIE SELF
```

实现标量的类应当提供以下方法：

```
TIESCALAR CLASS, LIST
FETCH SELF,
STORE SELF, VALUE
DESTROY SELF
UNTIE SELF
```

实现文件句柄的类应当提供以下方法：

```
TIEHANDLE CLASS, LIST
READ SELF, SCALAR, LENGTH, OFFSET
READLINE SELF
```



```

GETC SELF
WRITE SELF, SCALAR, LENGTH, OFFSET
PRINT SELF, LIST
PRINTF SELF, FORMAT, LIST
BINMODE SELF
EOF SELF
FILENO SELF
SEEK SELF, POSITION, WHENCE
TELL SELF
OPEN SELF, MODE, LIST
CLOSE SELF
DESTROY SELF
UNTIE SELF

```

并不是以上提到的所有方法都必须实现：`Tie::Hash`、`Tie::Array`、`Tie::Scalar`和`Tie::Handle`模块提供了一些基类，其中包含了合理的默认值。关于这些方法的详细讨论参见第14章。不同于`dbmopen`，`tie`函数不会为你用`use`或`require`加载一个模块，你要自己显式地声明这一点。另外参见`DB_File`和`Config`模块，可以了解一些有意思的`tie`实现。

## tied

```
tied VARIABLE
```

这个函数返回一个对象的引用，这个对象是`VARIABLE`中包含的标量、数组、散列或类型团的底层对象（`VARIABLE`是之前`tie`调用将变量绑定到一个包时返回的值）。如果`VARIABLE`没有绑定到一个包，它会返回一个未定义的值。所以，举例来说，可以使用：

```
ref tied %hash
```

查找你的散列绑定到哪个包（假设你忘记了）。

## time

```
time
```

这个函数返回自“纪元”（传统为1970年1月1日00:00:00 UTC）<sup>注22</sup>以来的平年秒数。返回值可以提供给`gmtime`和`localtime`，用于比较`stat`返回的文件修改时间和访问时间，还可以提供给`utime`。

```

$start = time();
system("some slow command");
$end = time();
if ($end - $start > 1) {
    say "Program started: ", scalar localtime($start);
    say "Program ended: ", scalar localtime($end);
}

```

---

注22：不要与“epic”（重大事件）混淆，那是指Unix的创建日（其他操作系统可能有不同的纪元，当然也会有不同的重大事件）。

```
}
```

要以更细的粒度（小于整数秒）来度量时间，可以使用`Time::HiRes`模块，Perl v5.8版本以后都提供了这个模块，而在此之前CPAN上已经有了这个模块。

## times

X
U

`times`

在列表上下文中，这个函数返回一个4元素的列表，提供这个进程及其已终止子进程的用户和系统CPU时间（时间单位为秒，可能是分数）。

```
($user, $system, $cuser, $csystem) = times();
printf "This pid and its kids have consumed %.3f seconds\n",
       $user + $system + $cuser + $csystem;
```

在标量上下文中，只返回用户时间。例如，要通过计时得到一段Perl代码的执行速度，可以写为：

```
$start = times();
...
$end = times();
printf "that took %.2f CPU seconds of user time\n",
       $end - $start;
```

## tr///

X
RO

`tr///`  
`y///`

这是变换（有时会错误地称为转换）操作符，它类似于Unix `sed`程序中的`y///`操作符，不过在所有人看来，这个操作符更好，参见第5章。

要使用一个只读值而不产生异常，可以使用`/r`修饰符，这个修饰符最早在v5.14中引入。

```
say "bookkeeper" =~ tr/boep/peob/r; # 打印"peekkoobor"
```

## truncate

\$!	X	X	X
ARG	ARG	T	U

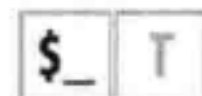
`truncate FILEHANDLE, LENGTH`  
`truncate EXPR, LENGTH`

这个函数将`FILEHANDLE`上打开的文件或名为`EXPR`的文件截断为指定的长度（字节数，而不是字符数）。如果你的系统上没有实现`ftruncate(2)`或一个等价的函数，这个函数会产生一个异常（如果你有充足的磁盘空间，完全可以通过复制文件的前面部分来完成文件的截断）。函数成功时返回`true`，否则返回`undef`。

如果`LENGTH`大于当前长度，行为是不确定的。不过，在传统Unix文件系统中，它会设置更大的文件长度，从而超过原来的文件末尾，内核会把中间的不可写的数据作为全0字节返回。

`FILEHANDLE`文件中的位置保持不变。对文件调用`truncate`之后，在写文件之前可能需要调用`seek`。

## uc



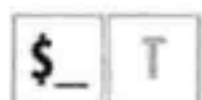
```
uc EXPR
uc
```

这个函数返回`EXPR`的一个大写版本。这是实现内插字符串中`\U`转义的内部函数。对于标题形式（`titlecase`），应当使用`ucfirst`。

不要用`uc`来完成不区分大小写的比较，在ASCII中你可能可以这么做，但是对于Unicode这会给出错误的答案。实际上，应当使用`fc`（`foldcase`）函数，可以利用CPAN `Unicode::CaseFold`模块，或者通过v5.16或以后版本中新增的`use feature "fc"`。另外参见第6章“张冠李戴”一节了解更多信息。

如果字符串没有Unicode语义（而且本地化环境模式未启用），128~256范围内的码点会被`uc`忽略，这一点可能很难猜测。`unicode_strings`特性可以确保这些码点也有Unicode语义。另外参见第6章。

## ucfirst



```
ucfirst EXPR
ucfirst
```

这个函数返回首字母采用标题形式（`titlecase`）而其他字符保持不变的`EXPR`版本。标题形式（`titlecase`）是Unicode的一种“说法”，表示首字母大写，而且后面是（或者希望有）小写字母，而不是大写字母。这样的例子有很多，如一个句子的首字母，人名的首字母，新闻标题的首字母，或者标题中的大多数单词。没有标题形式映射的字符会返回大写映射。这是实现双引号字符串中`\u`转义的内部函数。

例如，如果有人人在“flower”最前面使用了U+FB02 LATIN SMALL LIGATURE FL（即`"\x{FB02}ower"`），而你希望把它作为一个句子的第一个单词，它的标题形式映射为“Flower”，而不是“FLower”。不过，它的大写形式仍是“FLOWER”。

要设置一个字符串首字母大写，将它的第一个字符映射为相应的标题形式，其余字符为小写，可以使用以下代码：



```
ucfirst(substr($word, 0, 1)) . lc(substr($word, 1))
```

不要这样使用（除非你崇尚文化帝国主义）：

```
ucfirst lc $word
```

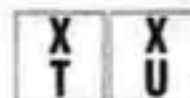
或"`\u\L$word`"，因为对于某些字符，这可能会生成一个不同而且不正确的答案。将一个字母变为小写，再取这个小写字符的标题形式，这与取原字符的标题形式可能并不相同。

因为标题形式只对一个字符串中的首字母（后面跟有小写字符）有意义，所以我们实在想不出任何理由需要对一个字符串中的每一个字符使用标题形式。不过万一你确实想这么做，可以如下实现：

```
$string =~ s/ ( (?= \p{CWT} ) \X ) /\u$1/gx;
```

这里我们使用的简写CWT属性的全名是`Changes_When_Titlecased=True`，不过要输入这个属性实在太长了，使用这个官方的正式缩写就可以。关于`unicode_strings`特性参见uc。

## umask



```
umask EXPR  
umask
```

这个函数使用`umask(2)`系统调用设置进程的掩码，并返回原来的掩码。掩码告诉操作系统创建一个新文件时（包括目录）哪些权限位要禁用。如果省略`EXPR`，这个函数只返回当前掩码。例如，要允许“user”位而禁用“other”位，可以使用以下代码：

```
umask((umask() & 077) | 7); # 不改变组权限位
```

记住掩码是一个数，通常按八进制给出；这不是一个八进制数的字符串。如果你得到的是一个字符串，参见`oct`。还要记住掩码位与正常的权限位是补码。

Unix权限`rwxr-x---`用3个3位一组的位集合（或者3个八进制位）表示：`0750`（前导0表示这是八进制，不算在3位中）。由于掩码的位是反转的，它表示的是禁用的权限位。你为`mkdir`或`sysopen`提供的权限（或“模式”）值会由`umask`修改，所以即使告诉`sysopen`以权限`0777`创建一个文件，如果掩码是`0022`，则会以权限`0755`来创建文件。如果`umask`是`0027`（组不可写；其他不可读、写或执行），为`sysopen`传入`MASK`为`0666`时，会创建一个模式为`0640`的文件（因为`0666 & ~0027`得到`0640`）。

下面给出一些建议：对于常规文件（`sysopen`中）可以提供创建模式为`0666`，对于目录（`mkdir`中）和可执行文件可以提供`0777`。这允许用户自由选择：如果他们希望文件得到保护，可以选择进程掩码为`022`、`027`或者甚至极端的掩码`077`。程序应当很少（甚至根本不要）做出策略性的决定，这些最好留给用户来做决定。对此有一个例外，写文件的程序应当保持私有，如邮件文件、Web浏览器cookie、`.rhosts`文件等。

如果你的系统上没有实现`umask(2)`，你要自己限制访问（也就是说，如果`(EXPR & 0700) > 0`），你要触发一个运行时异常）。如果未实现`umask(2)`，而且你未限制访问，这个函数会返回`undef`。

## undef



```
undef EXPR  
undef
```

`undef`是一个名字，我们用它表示一个称为“未定义值”的抽象。很合适地，返回未定义值的函数也是这个名字。我们很愿意将这二者混淆在一起<sup>注23</sup>。

巧合的是，如果提供一个实体的名字作为参数，`undef`函数也可以显式地取消这个实体的定义。如果指定了`EXPR`参数，这必须是一个左值。因此，只能对一个标量值、整个数组或散列、子例程名（使用`&`前缀）或类型团使用这个函数。与这个对象关联的所有存储空间都会恢复以便重用（尽管在大多数操作系统上都不是返回给系统）。对于大多数特殊变量，`undef`函数所做的处理可能也你期望的不同。对一个只读变量（如`$1`）使用这个函数会产生一个异常。

`undef`函数是一个一元操作符，而不是一个列表操作符，所以只能一次取消一个实体的定义。下面的例子将`undef`用作为一个一元操作符：

```
undef $foo;  
undef $bar{"blurfl"};      # 不同于delete $bar{"blurfl"};  
undef @ary;  
undef %hash;  
undef &mysub;  
undef *xyz;                # 撤销$xyz, @xyz, %xyz, &xyz, etc.
```

如果没有参数，`undef`就做未定义值使用：

```
select(undef, undef, undef, $naptime);  
  
return (wantarray ? () : undef) if $they_blew_it;  
return if $they_blew_it;    # 作用相同
```

可以使用`undef`作为列表赋值左边的一个占位符，在这种情况下，右边的相应值会被删除。除此以外，一般不会将`undef`作为一个左值。

```
($a, $b, undef, $c) = &foo;    # 忽略返回的第3个值
```

另外，不要试图将任何东西与`undef`比较，结果与你想象的不一样。它所做的是与`0`或`null`字符串比较。要测试一个值是否定义，应当使用`defined`函数或`//`操作符。

---

注23： 不过，Perl 6选择不混淆`undef`，但混淆了其他方面。

## unlink

`$_` `$_!` `X`  
`↑`

```
unlink LIST
unlink
```

这个函数删除一个文件列表<sup>注24</sup>。函数会返回成功删除的文件名个数。下面是一些例子：

```
$count = unlink("a", "b", "c");
unlink @goners;
unlink glob("*.orig");
```

`unlink`函数不会删除目录，除非你是超级用户，而且为Perl提供了`-U`命令行选项。即使这些条件满足，也要当心删除一个目录可能会对你的文件系统造成严重的危害。应当使用`rmdir`。

下面是一个简单的`rm`命令，这里提供了非常简单的错误检查：

```
#!/usr/bin/perl
@cannot = grep {not unlink} @ARGV;
die "$0: could not unlink all of @cannot" if @cannot;
```

## unpack

`$@`

```
unpack TEMPLATE, EXPR
```

这个函数与`pack`的工作正好相反：它根据`TEMPLATE`将表示一个数据结构的字符串（`EXPR`）扩展为一个值列表，并返回这些值。`pack`和`unpack`的模板见第26章的介绍。

## unshift

`X`  
`ARG`

```
unshift ARRAY, LIST
```

这个函数与`shift`的工作正好相反（或者与`push`相反，这取决于你如何看待）。它将`LIST`追加到数组前面，并返回数组中新的元素个数：

```
unshift(@ARGV, "-e", $cmd) unless $ARGV[0] =~ /^-/;
```

注意`LIST`会整体追加，而不是一次追加一个元素，所以追加的元素仍保持原来的顺序。可以使用`reverse`来逆转顺序。

从v5.14开始，`unshift`可以取一个未祝福的数组的引用，它会自动解引用。`unshift`的这个方面还是试验性的。具体的行为在Perl将来的版本中有可能改变。

---

注24：实际上，在一个POSIX文件系统上，它会删除指示实际文件的目录项（文件名）。由于文件可能被多个目录引用（链接），在最后一个文件引用删除之前，这个文件不会删除。



## untie

`untie VARIABLE`

断开`VARIABLE`中包含的变量或类型团与它绑定的包之间的绑定关系。另外参见`tie`和第14章，不过特别要参考“一个解除绑定小陷阱”一节。

## use

`$!` `$@`

```
use MODULE VERSION LIST
use MODULE VERSION ()
use MODULE VERSION
use MODULE LIST
use MODULE ()
use MODULE
use VERSION
```

`use`声明会加载一个模块（如果之前没有加载），并从指定模块将子例程和变量导入到当前包（从技术上讲，它会从指定模块向当前包导入一些语义，通常这是通过对包中的某些子例程或变量名建立别名做到的）。大多数`use`声明都如下所示：

```
use MODULE LIST;
```

这就等价于：

```
BEGIN { require MODULE; import MODULE LIST }
```

`BEGIN`强制`require`和`import`在编译时发生。`require`确保如果模块尚未加载到内存中，则将其加载到内存中。`import`并不是一个内置函数，这只是`MODULE`包中的一个普通的类方法调用，告诉该模块将特性列表放回到当前包中。这个模块可以用它喜欢的任何方式实现其导入方法，不过大多数模块只是通过从`Exporter`继承（在`Exporter`模块中定义）来派生其导入方法。参见第11章和`Exporter`模块来了解更多信息。如果没有找到任何`import`方法，会跳过这个调用，不会有任何警告。

如果你不希望改变你的命名空间，可以显式地提供一个空列表：

```
use MODULE ();
```

这就等价于：

```
BEGIN { require MODULE }
```

如果`use`的第一个参数是一个版本号（如`v5.12.3`），当前执行的Perl版本应当至少是指定的这个版本。如果当前Perl版本小于`VERSION`，会打印一个错误消息，Perl会立即退出。如果要加载一个依赖于更新版本的库模块，在此之前需要检查当前的Perl版本，这个函数就很有用，因为有时我们必须“破坏”Perl较早版本的一些不好的特性（我们尽量做到不会矫枉过正。事实上，需要破坏的特性很多，而我们实际破坏的还远没有那么多）。

谈到不破坏特性，Perl还接受以下形式的千禧年版本号：

```
use 5.005_03;
```

不过，为了更好地与行业标准看齐，进入新千年发布的所有Perl版本都接受（而且我们希望看到）这种3段形式：

```
use 5.12.0; # 这表示版本5，子版本12，补丁级别0。
use v5.12.0; # 同上
use v5.12; # 同上，不过记得一定要加v!
use 5.012; # 同上，为了与更老的Perl版本兼容
use 5.12; # 不正确!
```

如果MODULE后面提供了VERSION参数，use会调用类MODULE中的VERSION方法，并以给定的VERSION作为参数。需要说明，VERSION后面没有逗号！如果给定的版本大于变量\$Module::VERSION的值，默认VERSION方法（这是从UNIVERSAL类继承来的）会发出警告。

另外从生产版本v5.10开始，use VERSION还会加载feature pragma，并启用所请求版本中所有可能的特性。参见第29章中“feature”一节。类似地，如果指定的Perl版本是生产版本v5.12或更高版本，通过use strict可以在词法作用域中启用结构（除非没有真正加载strict.pm文件）。

因为use提供了一个开放的接口，pragmas（编译器指令）也通过模块实现。当前实现的pragma包括：

```
use autouse "Carp" => qw(carp croak);
use bigint;
use constant PI => 4 * atan2(1,1);
use diagnostics;
use integer;
use lib "/opt/projects/spectre/lib";
use locale;
use sigtrap qw(die INT QUIT);
use sort qw(stable _quicksort _mergesort);
use strict qw(subs vars refs);
use threads;
use warnings qw(numeric uninitialized);
use warnings qw(FATAL all);
```

这些实现pragma的模块中，很多都会向当前词法作用域导入语义（这与普通的模块不同，它们只向当前包导入符号，与当前词法作用域没有太大关系，只是在编译词法作用域时会考虑到这个包。也就是说……嗯，算了，参见第11章吧）。

由于use在编译时发生，它不会考虑所编译的代码的正常流控制。具体来讲，如果在一个条件的false分支中放入一个use，它也会得到处理。如果一个模块或pragma需要有条件地加载，可以使用if pragma来做到：

```
use if $] < 5.008, "utf8";
```

```
use if WANT_WARNINGS, warnings => qw(all);
```

还有一个对应的声明`no`，它会“取消导入”之前由`use`导入的那些不太重要的特性：

```
no integer;
no strict qw(refs);
no warnings qw(deprecated);
```

使用`no VERSION`形式的`no`时要特别当心。它只是用来断言当前运行的Perl版本比其参数指定的版本要早，而不会取消`use VERSION`启用的某些特性。参见第29章，其中给出了标准`pragma`的一个列表。

## utime



utime LIST

这个函数改变一个文件列表中各个文件的访问和修改时间。列表中的前两个元素必须是数值的访问时间和修改时间（按照这个顺序）。这个函数返回成功改变的文件个数。各个文件的inode改变时间设置为当前时间。下面是一个例子，`touch`命令将文件的修改日期设置为将来的某个月（假设你是这个文件的所有者）：

```
#!/usr/bin/perl
# montouch 一推迟发布的文件，当前时间+ 1个月
$day = 24 * 60 * 60;          # 24小时的秒数
$later = time() + 30 * $day;  # 一个月大约30天
utime $later, $later, @ARGV;
```

下面是一个更复杂的类`touch`命令，加了一点错误检查：

```
#!/usr/bin/perl
# montouch - 推迟发布的文件，当前时间+ 1个月
$later = time() + 30 * 24 * 60 * 60;
@cannot = grep {not utime $later, $later, $_} @ARGV;
die "$0: Could not touch @cannot." if @cannot;
```

要从现有文件读取时间，可以使用`stat`，然后通过`localtime`或`gmtime`传递适当的字段进行打印。

在NFS下，这会使用NFS服务器的时间，而不是本地机器的时间。如果存在时间同步问题，NFS服务器和本地机器会有不同的时间。正常情况下，Unix `touch(1)`命令实际上会使用这种形式，而不是第一个例子中所示的形式。

如果为前两个元素的其中一个传入`undef`，这等价于传入一个0，所以这与两个都是`undef`时描述的效果不同。这种情况下还会触发一个未初始化警告。

在支持`futimes(2)`的系统上，可以在文件之间传递文件句柄。在不支持`futimes(2)`系统调用



的系统上，传递文件句柄会产生一个异常。为了便于识别，文件句柄必须作为类型团或类型团引用传递，裸字会认为是文件名。

```
utime($then, $then, $then, *SOME_HANDLE);
```

## values



```
values HASH  
values ARRAY
```

这个函数返回一个列表，包含指定散列中的所有值。这些值会以一种看上去随机的顺序返回，不过这个顺序与keys或each函数在这个散列上生成的顺序相同。奇怪的是，要按值对一个散列排序，通常需要使用keys函数，所以参见keys下的例子。

可以使用这个函数修改一个散列的值，因为返回的列表包含值的别名，而不是副本（在较早的版本中，为此需要使用一个散列片段）。

```
for (@hash{keys %hash}) { s/foo/bar/g }      # 老方法  
for (values %hash) { s/foo/bar/g }           # 现在改变值的方法
```

如果散列绑定到一个超大的DBM文件，对这样一个散列使用values时会生成一个超大的列表，这会导致一个庞大的进程。你可能更愿意使用each函数，它会逐个元素地迭代处理散列项，而不是把它们一次全部吞入到一个庞大的列表中。

## vec



```
vec EXPR, OFFSET, BITS
```

vec函数提供了无符号整数列表的压缩存储。这些整数尽可能紧密地压缩在一个正常的Perl字符串中。EXPR中的字符串会处理为一个位串，由任意多个元素组成，这取决于字符串的长度。

OFFSET指定你感兴趣的特定元素的索引。读写这个元素的语法是一样的，因为vec要根据在左值上下文还是右值上下文中使用这个元素来确定是存储还是返回元素的值。

BITS指定各个元素的宽度（位数），这必须是2的幂：1，2，4，8，16或32（一些平台上还可以是64）。如果使用其他值，会产生一个异常。因此每个元素可以包含范围在 $0..(2^{BITS})-1$ 之间的一个整数。如果BITS比较小，会把尽可能多的元素压缩到各个字节中。BITS为1时，每个字节有8个元素。BITS为2时，每个字节有4个元素。如果BITS为4，每个字节有2个元素（通常称为半字节），依此类推。大于一个字节的整数会按大端字节顺序存储。

通过将无符号整数逐个地赋给vec函数，可以把这个无符号整数列表存储到一个标量变量

中。（如果`EXPR`不是一个合法的左值，会产生一个异常）。在下面的例子中，元素分别有4位：

```
$bitstring = "";
$offset = 0;

for my $num (0, 5, 5, 6, 2, 7, 12, 6) {
    vec($bitstring, $offset++, 4) = $num;
}
```

如果元素超出所写字符串的末尾，Perl首先会用足够多的0字节扩展该字符串。

接下来可以通过指定正确的`OFFSET`来获取存储在标量变量中的向量。

```
$num_elements = length($bitstring)*2; # 每个字节2个元素

for my $offset (0 .. $num_elements-1) {
    say vec($bitstring, $offset, 4);
}
```

如果所选择的元素超出字符串末尾，会返回一个值0。

用`vec`创建的字符串也可以用逻辑操作符`|`，`&`，`^`和`~`处理。两个操作数都是字符串时，这些操作符会假定需要完成一个位串操作。参见第3章“位操作符”一节中的例子。

如果`BITS == 1`，可以创建一个位串，将一个位序列存储在一个标量中。`vec($bitstring,0,1)`要保证在这个串的第一个字节的最低位上。

```
@bits = (0,0,1,0, 1,0,1,0, 1,1,0,0, 0,0,1,0);

$bitstring = "";
$offset = 0;

for my $bit (@bits) {
    vec($bitstring, $offset++, 1) = $bit;
}

say $bitstring; # "TC", ie. '0x54', '0x43'
```

通过为`pack`或`unpack`提供一个“`b*`”模板，位串可以与I/O字符串来回转换。或者，可以结合“`b*`”模板使用`pack`，从一个由1和0组成的字符串创建一个位串。其顺序与`vec`期望的顺序一致。

```
$bitstring = pack "b*", join(q(), @bits);
say $bitstring; # "TC",与前面相同
```

`unpack`可以从位串抽取出0和1的列表。

```
@bits = split(//, unpack("b*", $bitstring));
say "@bits"; # 0 0 1 0 1 0 1 0 1 1 0 0 0 0 1 0
```

如果你知道具体长度（位数），可以用这个具体长度取代“`*`”。

参见select，从中可以看到使用由vec生成的位图的更多例子。另外参见pack和unpack，来了解如何在更高层次处理二进制数据。

## wait

\$!	\$?	X
		U

wait

这个函数等待子进程终止，并返回已停止进程的PID，或者如果没有子进程（或者，在某些系统上，子进程可能会自动俘获），则返回-1。这个状态返回到\$?中，见system下的介绍。如果有僵尸子进程，就要调用这个函数，或者调用waitpid。

如果你想要一个子进程，但是使用wait没能找到这个子进程，这可能是已经调用了system，关闭了管道，或者在fork和wait之间使用了反引号。这些构造也会完成一个wait(2)，可能已经俘获了你的子进程。可以使用waitpid避免这个问题。

## waitpid

\$!	\$?	X
		U

waitpid PID, FLAGS

这个函数等待某个特定的子进程终止，并在该进程死亡时返回PID，如果没有子进程则返回-1，或者如果FLAGS指定非阻塞而且进程还没有终止，就会返回0。所有死亡进程的状态都返回到\$?中，见system下的介绍。要得到合法的标志值，可能需要从POSIX模块导入“:sys\_wait\_h”导入标记组。下面给出一个例子，这里会以非阻塞方式等待所有未结束的僵尸进程。

```
use POSIX ":sys_wait_h";
do {
    $kid = waitpid(-1,&WNOHANG);
} until $kid == -1;
```

在既没有实现waitpid(2)也没有实现wait4(2)系统调用的系统上，FLAGS只能指定为0。换句话说，可以在这里等待一个特定的PID，不过在非阻塞模式下不能等待。

有些系统上，返回值-1可能还表示子进程会自动俘获，这可能是因为你设置了\$SIG{CHLD} = "IGNORE"。

## wantarray

wantarray

如果当前执行的子进程的上下文在查找一个列表值，这个函数返回true，否则返回false。如果调用上下文在查找一个标量，则返回一个已定义>false值("")。如果调用上下文没有寻找任何东西，也就是说，如果它在void上下文中，则返回未定义>false值(undef)。



下面是一些典型用法的例子：

```
return unless defined wantarray;      # 不要考虑做更多事情
my @a = complex_calculation();
return wantarray ? @a : \@a;
```

参见`caller`。这个函数实际上应该名为“`wantlist`”，但是为这个函数取名时，那时列表上下文还被叫做数组上下文。

## warn

\$!

```
warn LIST
warn
```

这个函数会生成一个错误消息，就像`die`一样，向`STDERR`打印`LIST`，不过它不会退出，也不会抛出一个异常。例如：

```
warn "Debug enabled" if $debug;
```

如果`LIST`为空而且`$@`已经包含一个值（通常来自前一个`eval`），会在`STDERR`上将字符串“`\t...caught`”追加到`$@`后面（这类似于`die`传播错误的方式，不过`warn`不会传播[重新产生]异常）。如果所提供的消息字符串为空，会使用消息“`Warning: Some thing's wrong`”。

与`die`类似，如果所提供的字符串不是以一个换行符结尾，会自动追加文件和行号信息。`warn`函数与Perl的`-w`命令行选项没有任何关系，但是可以结合使用，例如，你希望模拟内置函数时：

```
warn "Something wicked\n" if $^W;
```

如果安装了一个`$SIG{__WARN__}`处理器，则不会打印任何消息。要由这个处理器负责适当地处理消息。你可能想把警告提升为异常：

```
local $SIG{__WARN__} = sub {
    my $msg = shift;
    die $msg if $msg =~ /isn't numeric/;
};
```

因此大多数处理器必须做好安排来显示不准备处理的警告，可能在处理器中再次调用`warn`。这样非常安全。不会产生一个无限的循环，因为`__WARN__`钩子不会从`__WARN__`钩子内部调用。这个行为与`$SIG{__DIE__}`处理器的做法稍有区别（它不会抑制错误文本，但是可以再次调用`die`来改变消息文本）。

通过使用`__WARN__`处理器，提供了一种强大的方式来抑制所有警告，甚至是那些所谓的强制警告。有时你需要把这个代码包装在一个`BEGIN{}`块中，使它能在编译时发生：

```

# 清除*所有*编译时警告
BEGIN { $SIG{ _ _WARN_ _ } = sub { warn $_[0] if $DOWARN } }
my $foo = 10;
my $foo = 20;          # 没有关于重复my $foo的警告,
                        # 不过, 这是你要求的!

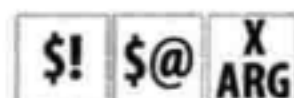
# 在此之前没有编译时或运行时警告
$DOWARN = 1;           # *不是*一个内置变量

# 在此之后启用运行时警告
warn "\$foo is alive and $foo!";      # 确实会显示警告

```

对于警告的词法作用域控制, 参见warnings pragma。另外参见Carp模块的carp和cluck函数, 了解生成警告消息的其他方法。

## write



```

write FILEHANDLE
write

```

这个函数使用与指定文件句柄关联的格式向这个文件句柄写一个格式化记录（可能多行），参见第26章“格式变量”一节。默认地，与文件句柄关联的格式与文件句柄同名。不过，在用select选择了句柄之后可以通过修改变量\$~改变这个文件句柄的格式：

```

$old_fh = select(HANDLE);
$~ = "NEWNAME";
select($old_fh);

```

或者可以写为：

```

use IO::Handle;
HANDLE->format_name("NEWNAME");

```

由于格式要放在包命名空间中，如果format在另外一个包中声明，还必须用包名完全限定这个格式名：

```

$~ = "OtherPack::NEWNAME";

```

这会自动完成表格顶端处理。如果当前页没有足够的空间放置这个格式化记录，会写一个换页进纸符，这会对新的页眉使用一个特殊的页面顶端格式，然后写这个记录。当前页上余下的行数置于变量\$-中，这可以设置为0，强制下一次调用write时写一个新页（可能要先用select选择文件句柄）。默认地，页面顶端格式的名字就是文件句柄名再加上“\_TOP”，不过文件句柄的格式可能会改变，可以在选择句柄后修改\$^变量，或者使用以下代码：

```

use IO::Handle;
HANDLE->format_top_name("NEWNAME_TOP");

```

如果FILEHANDLE未指定，会输出到当前的默认输出文件句柄，初始的默认输出文件句柄

是STDOUT，但是可以使用单参数形式的select操作符改变这个默认输出文件句柄。如果FILEHANDLE是一个表达式，则会在运行时计算这个表达式来确定实际的FILEHANDLE。

如果指定的format或当前页面顶端format不存在，会产生一个异常。

write函数并不是read的逆操作。真是很遗憾。简单的字符串输出要使用print。如果因为希望绕过标准I/O才找到了这个函数，建议你参见syswrite。

y//

y///

变换（历史上称为转换，不过不太准确）操作符也称为tr///。参见第5章。



# 标准Perl库

标准Perl发行版不只包含运行脚本的Perl可执行程序。它还包括数百个包含可重用代码的模块，我们称之为标准Perl库。因为这些标准模块可以在任何地方使用，假如你在程序中使用了某个模块，那么只要安装了Perl就可以运行你的程序，而无需任何额外的安装步骤。

不过需要说明，并不是有Perl的地方都有标准Perl库（Standard Perl Library, SPL）。因为Perl支持很多不同的平台，你可能正好碰到一些开发商改变了Perl。有些开发商可能会增强Perl，增加额外的模块或工具。有些可能会更新一些模块，从而更适用于他们的平台（我们希望这些开发商能上传他们的补丁）。不过，另外一些开发商可能会删除一部分模块。如果你发现缺少标准库的某些部分，可以找你的开发商，或者安装你自己的Perl。

在这本书的前几版中，我们在标准库一章列出了每一个模块，并分别对各个模块做了一点介绍。这一版中我们去掉了这个内容，而会告诉你如何自己去了解标准库。第29章还会介绍pragma。

## 标准库术语

下面简要回顾我们一直都在谈到的术语。我们以及整个Perl社区对于这些术语的使用并不严格，因为这些概念往往是重叠或并存的，不过有时准确性很重要。

### 命名空间（namespace）

命名空间是一个维护名字的地方，使这些名字不会与其他命名空间的名字混淆。这样一来，你只需要考虑一个更简单的问题：这些命名空间本身不能混淆。有两种方法避免命名空间彼此混淆：可以为它们提供唯一的名字，或者为它们指定唯一的位置。这两种方法在Perl中都是允许的：有名字的命名空间称为包，没有名字的命名空

间称为词法作用域 (lexical scopes)。由于词法作用域不能大于一个文件，另外由于标准模块大小 (至少) 为一个文件，所以如果要在文件以外使用模块，模块的所有接口都必须使用有名字的命名空间 (包)。

### 包 (package)

包是Perl声明有名字的命名空间的标准机制。这是一种将相关函数和变量组织在一起的简单机制。只有两个目录可以包含名为`Amelia`的 (不同) 文件，一个Perl程序的两个不同部分可以分别有自己的`$Amelia`变量或`&Amelia`函数。尽管这些函数或变量看起来名字相同，但不同命名空间中的这些名字会由`package`声明来管理。包名用来标识模块和类，如第11章和第12章所述。

### 库 (library)

很遗憾，库这个词在Perl文化中有多种含义。如今，我们通常使用这个词表示系统上安装的所有Perl模块。

历史上，Perl库是一个包含一组子例程的文件，这些子例程有某种共同的用途。这样一个文件的扩展名通常是`.pl`<sup>注1</sup>，这是“perl库” (Perl library) 的简写。用`do FILE`或用`require`加载的Perl代码会使用这个扩展名。尽管这不是一个完整的模块，但库文件通常声明自己在一个包中，使得相关的变量和子例程可以放在一起维护，而不会与程序中的其他变量冲突。并没有强制性的扩展名，除了`.pl`外有时也会有其他的扩展名，这一章后面还会解释。这些简单的非结构化的库文件很大程度上已经被模块的概念所取代。

### 模块 (module)

Perl模块是一个符合某些特定约定的库文件，允许通过一个`use`声明在编译时加载实现该模块的一个或多个文件。模块文件名总是以`.pm`结尾，因为`use`声明认为模块都有`.pm`扩展名。`use`声明还会把包分隔符`::`转换为你的目录分隔符，使得Perl库中的目录结构与你的包结构一致。第11章介绍了如何创建你自己的Perl模块。

### 类 (class)

类就是一个模块，它为与模块包名相关的对象实现了方法。如果对面向对象模块感兴趣，参见第12章。

### `pragma`

`pragma`是一个调整Perl内部开关的特殊模块，通常用于改变编译器以何种方式进行解释，以及增加特殊的行为。参见第29章介绍的标准库`pragma`。

### 扩展 (extension)

扩展是一个Perl模块，除了加载一个`.pm`文件外，还可以加载用C或C++实现模块语义的共享库。

---

注1： 没错，Perl程序也使用这个扩展名。我们猜想也许你习惯这样，或者可能是你的操作系统要求你建立有趣的图标。



## 程序 (program)

Perl程序设计为将作为独立实体运行的代码。如果你不希望别人期待它有太多功能也可以称之为脚本 (script)，如果很庞大很复杂，也可以称之为应用 (application)，如果它的调用者不关心它是用什么语言写的，可以称之为可执行程序 (executable)，或者如果它花费了资金，还可以称为企业解决方案 (enterprise solution)。Perl程序可以作为源代码、字节码或原生机器码存在。只要能够从命令行运行，我们就称之为一个程序。

## 发行版本 (distribution)

发行版本是脚本、库或模块以及一个测试套件、文档和安装脚本的归档。人们谈到“从CPAN获取一个模块”时，实际上就是指一个发行版本。参见第19章。

# Perl库之旅

如果花些功夫来熟悉标准库，你会节省大量时间，因为这样一来你就不用凡事都从头做起。不过要注意，Perl库包含了各种各样的资源。尽管有些库可能非常有帮助，但另外一些可能与你的需要完全不相关。例如，如果你只是用100%的纯Perl写代码，支持C和C++扩展的那些动态加载的模块对你就没有帮助。

Perl希望在其库“包含”路径@INC中查找库模块。这个数组指定了一个有序的目录列表，使用关键字do、require或use加载某些库代码时，Perl将在这些目录中搜索。通过调用Perl并提供-V开关，可以很容易地列出这些目录，这会提供非常详细的版本 (Very Verbose Version) 信息，或者使用以下简单的代码也可以得到这些目录：

```
% Perl-le "print for @INC"
/usr/local/lib/perl5/site_perl/5.14.2/darwin-2level
/usr/local/lib/perl5/site_perl/5.14.2
/usr/local/lib/perl5/5.14.2/darwin-2level
/usr/local/lib/perl5/5.14.2
.
```

这只是可能的输出的一个例子。每个Perl安装都会使用它自己的路径。重点是，尽管取决于开发商和你的网站安装策略的不同，Perl库的内容可能会有所不同，但是完全可以信任随Perl安装的所有标准库。前面显示的目录来自一个手动安装的Perl。同一个系统上安装的另一个Perl可能会给出不同的答案：

```
% /usr/bin/Perl-le "print for @INC"
/Library/Perl/5.12/darwin-thread-multi-2level
/Library/Perl/5.12
/Network/Library/Perl/5.12/darwin-thread-multi-2level
/Network/Library/Perl/5.12
/Library/Perl/Updates/5.12.3
/System/Library/Perl/5.12/darwin-thread-multi-2level
/System/Library/Perl/5.12
```



```
/System/Library/Perl/Extras/5.12/darwin-thread-multi-2level
/System/Library/Perl/Extras/5.12
.
```

这个输出（来自Mac OS X.7）有很大不同，展示了不同的模块位置。标准库在/System下，但是更新放在靠近@INC前面的一个目录中。更新Mac OS X时，并不是覆盖标准库，它会将更新放在另外一个开发商特定的目录中。你安装的模块会放在/Library下，这样一来操作系统更新就不会覆盖你做的修改。除非你在安装模块时另有要求（参见第19章），否则都会如上所述放在相应的位置上。

如果查看这个Perl的目录，你可能会发现一些相同的模块但有不同的版本，另外还有额外的一些模块。有些开发商会对标准库应用他们自己的补丁。可能会更新版本，也可能不更新。如果你认为你的Perl与其他人的不一样，可以检查它们到底是不是同一个东西。

perldoc命令的-l会报告一个模块的位置：

```
% perldoc -l MODULE
/usr/local/lib/perl5/site_perl/5.14.2/MODULE
```

在一个程序中，%INC变量会跟踪已经加载了哪些模块，以及在哪里找到这些模块。键是转换为文件路径的命名空间，如Unicode/UCD.pm，值是模块的路径。有关的更多详细信息参见第25章。

这会带来模块加载的一个问题。Perl使用它在@INC中找到的第一个匹配的文件。如果以后还有新的版本，它不会查找最新或最好的版本，也没有一种好的办法能让Perl继续查找，除非在一个代码引用中（放在@INC最前面）重新实现整个过程<sup>注2</sup>，或者创建一个新的库路径，链接到所有其他目录中模块的“最佳”版本。这很麻烦，很费工夫。例如，如果有人设置了PERL5LIB，你应该选择在那里找到的版本而不是查找更新的目录吗？

## 名单

在本书以前的版本中，我们提供了一个列表，列出了标准库中的所有模块，不过这本书已经太厚了，实在不能再用几十页介绍这个内容，特别是考虑到你可能只想查看perlmodlib来查找你的Perl版本的相应列表。如果你不喜欢这样，可以自行建立这个列表，查找所有.pm文件，然后抽取=head1 NAME后面的非空行：

```
use v5.10;

use File::Find;

my %names;
my $wanted = sub {
    return unless /\.pm$/;

```

---

注2： inc::latest模拟提供了一个代码引用可以完成这个工作。

```

open(my $fh, "<", $File::Find::name)
|| die "can't open $File::Find::name: $!";
OUTER: while( <$fh> ) {
    next unless /\A =head1 \s+ NAME/x;
    while( <$fh> ) {
        next if /\A \s* \z/x;
        / (?<name>\S+) \s* -+ \s* (?<desc>.*) /x;
        $names{ $+{name} } = $+{desc};
        last OUTER;
    }
}
};

find($wanted, @INC);

for my $name (sort keys %names) {
    printf "%-25s - %s\n", $name, $names{$name};
}

```

对于v5.14, 这会找到大约500个命名空间:

AnyDBM_File	- provide framework for multiple DBMs
App::Cpan	- easily interact with CPAN from
	- the command line
App::Prove	- Implements the C<prove> command.
... many others ...	
warnings	- Perl pragma to control optional warnings
warnings::register	- warnings import function
writemain	- write the C code for perl main.c

还有一种办法。Module::CoreList模块（标准库中的一个模块）知道哪个Perl提供了什么。它的corelist模块是接口。要查找它知道的版本，可以使用-v开关：

```

% corelist -v
5
5.000
5.001
5.002
...
v5.14.0
v5.14.1

```

对于某个版本，-v会报告这个Perl版本提供的所有模块和版本：

```

% corelist -v 5.14.1
The following modules were in Perl 5.14.1 CORE
AnyDBM_File          1.00
App::Cpan             1.5701
App::Prove            3.23
...many more...
version              0.88
vmsish               1.02
warnings             1.12
warnings::register    1.02

```

还可以用`-a`报告一个模块的历史：

```
% corelist -a Archive::Extract
Archive::Extract was first released with Perl v5.9.5
v5.9.5      0.22_01
v5.10.0     0.24
v5.10.1     0.34
...
v5.14.0     0.48
v5.14.1     0.48
```

如果你想知道包含该模块的最早的Perl版本，不要使用任何开关：

```
% corelist Module::CoreList
Module::CoreList was first released with Perl v5.8.9
```

`-d`开关会报告打算包含该模块的最早的Perl版本。例如，在v5.9.2之前Module::CoreList没有加入Perl标准库：

```
% corelist -d Module::CoreList
Module::CoreList was first released with Perl v5.9.2
```

`-d`表示“日期”（date），所以不要弄混了。Perl v5.9.2是v5.8.9之前临时发布的，这使得这个结果看起来有些奇怪。

## 标准Perl库的未来

对于标准Perl库的未来，有两种想法一直在斗争。一种是标准Perl库中要有尽可能多的模块，人们可以使用他们喜欢的模块创建应用，并且能很容易地发布这些应用而不需要别人安装额外的模块。另一种想法是希望有一个最小的发行版本，其中只有适当数量的模块，允许以后从CPAN安装其他模块。

这两种想法都各有优点。更大的库对用户来说很有好处。一旦有了Perl，他们不用再麻烦他们的系统管理员和律师允许他们安装另外的模块。比较小的库对Perl5 Porters则更有利，他们可以少花功夫处理模块，可以把更多的时间用在处理其他任务上。

有些模块有双重性（dual-lived），这是指它们有两条开发轨道。一个是Perl存储库本身，另一个是在CPAN上。这允许模块比Perl发行周期更快地对问题做出补丁。发行一个新版本的Perl时，维护人员会把CPAN版本的修改合并到Perl源代码中。有时，Perl存储库中的版本会先得到修正。在这种情况下，CPAN开发人员会在有空时合并这些改变。

多年以来，这个过程很繁琐，因为CPAN版本和标准库版本的布局有很大不同，使得这种合并成为一个很麻烦、很难自动完成的过程。除了对模块加补丁，维护人员还必须将测试合并到Perl测试套件中，在适当的地方放置辅助文件等。这是一件任何人都不想做的事情。现在的趋势是把CPAN发行版本完整地放在Perl存储库中它自己的目录下，从而能很容



易地加入对模块的修改。等你读到这本书时，这一点可能已经完成了。维护双重性模块的工作在过去几年已经得到很大改善。这使得开发商可以很容易地在其定制的Perl发行版本中包含额外的模块。

## 巡礼

查看@INC及其子目录警方，你会发现安装了不同类型的文件。大多数文件名都以`.pm`结尾，不过有些以`.pl`，`.ph`，`.al`或`.so`结尾。你最感兴趣的是第一种，因为后缀`.pm`表示这个文件是一个Perl模块。有关的更多内容稍后介绍。

以`.pl`结尾的文件不多，这些是我们之前提到过的较老的Perl库。包含这些文件是为了与20世纪80年代和20世纪90年代初期的古老的Perl版本兼容。正因如此，很早以前（比如说1990年）能工作的Perl代码完全还可以继续很好地工作，即使你已经安装了一个现代版本的Perl。编写利用了标准Perl库的新代码时，一定要尽可能选择使用`.pm`版本而不是`.pl`。这是因为这些模块不会像很多老的`.pl`文件那样污染你的命名空间。不过，随着Perl的发展，Perl 5 Porters已经删除了这样一些文件，可能把有关的任务委托给模块，或者建议你到CPAN获取这些模块。

关于`.pl`扩展名的使用有一点需要说明：它表示Perl库而不是Perl程序。尽管`.pl`有时用于标识Web服务器上的Perl程序（这些服务器可能需要对可执行程序 and 相同目录中的静态内容加以区分），或者一些系统会把文件与某个程序相关联来打开这个文件，我们建议使用后缀`.plx`来指示一个可执行的Perl程序（对于根据文件扩展名选择解释器的操作系统，我们也有同样的建议），或者根本不使用任何扩展名，因为Perl并不关心你把它叫什么。只要其中的文本是一个Perl程序，它会很高兴地运行`hello.rb`<sup>注3</sup>。

有`.al`扩展名的文件都是较大模块中的小段代码，使用其父`.pm`文件时，它们会自动加载。如果使用随Perl（而且没有使用Perl的`-A`标志）提供的标准`h2xs`工具（参见第19章）构建你的模块布局，`make install`过程会使用`AutoLoader`模块（因此有`a`和`l`）来为你创建这些小`.al`文件。

`.ph`文件由标准`h2ph`程序建立，这是一个有点古老但有时仍很必要的工具，它会尽其所能把C预处理器指令转换为Perl。得到的`.ph`文件包含底层函数（如`ioctl`、`fcntl`或`syscall`）有时需要的常量（如今，这些常量大多可以在类似POSIX，`Errno`，`Fcntl`或`Socket`模块等标准模块中更方便、更可移植地得到）。另外参见`perlinstall`了解如何安装这些可选但有时很重要的组件。

你可能碰到的最后一种文件扩展名是`.so`（或者你的系统对共享库使用的任何扩展名）。这些`.so`文件是扩展模块依赖于平台的部分。原来用C或C++编写，这些模块已经编译为动态

---

注3： Parrot解释器的目标之一就是能够加载和运行一个`hello.rb`，即使它包含Ruby代码。

可重定位的对象代码。不过，最终用户不需要知道它们的存在，因为模块接口隐藏了这些代码。用户代码声明`require MODULE`或`use MODULE`时，Perl会加载`Module.pm`并执行，使模块引入所有必要的部分，如`Module.so`或所有自动加载的`.al`组件。实际上，模块可以加载它想加载的任何东西，包括582个其他模块，所有那些模块还可以分别加载另外的582模块。只要它愿意，可以下载整个CPAN，以及最近两年的`freshmeat.net`归档。

模块不只是Perl中的一个静态代码块。它是一个活动的代理，可以确定如何代表你实现一个接口。它可能遵循所有标准约定，或者可能不遵循。可以做任何处理来调整其余程序的含义，甚至包括将你的其余程序转换为SPITBOL。只要有明确的文档，这种技巧都认为是完全合理的。使用这样一个Perl模块时，就说明接受了它的契约，而不是Perl所写的标准契约。

所以最好仔细阅读要它的合约条款。

# 实现Pragma的模块

*pragma*是一种会影响程序编译阶段的特殊模块。一些实现Pragma的模块（或简写为*pragmata*，或可以进一步简写为*pragma*）还可能影响你的程序的执行阶段。可以把它想成是编译器的提示。因为这些提示必须在编译时可见，所以由*use*或*no*调用时才有效，否则等*require*或*do*运行时，编译早已经结束了。

按约定，*pragma*名都用小写，因为小写模块名是为Perl发行版本自身保留的。编写你自己的模块时，模块名中至少要使用一个大写或标题形式字符，以免与*pragma*名冲突。

不同于常规的模块，大多数*pragma*会把其效果限制在从调用它们开始的最内层外围块的其余部分。换句话说，它们限制在词法作用域，就像*my*变量一样。正常情况下，外部块的词法作用域会涵盖其中内嵌的所有内部块，但是一个内部块可以通过使用*no*语句取消外部块中的一个词法作用域*pragma*：

```
use strict;
use integer;
{
    no strict "refs";      # 允许符号引用
    no integer;            # 恢复浮点数运算
    # ....
}
```

相比Perl提供的其他模块，*pragma*构成了Perl编译环境的一个非常重要而且不可缺少的部分。如果你不知道如何为编译器传递提示，就很难很好地使用编译器，所以我们再花点工夫来介绍*pragma*。

另外要注意的一点是，我们通常使用*pragma*来建立特性原型，这些特性以后会实现为“真正”的语法。所以在一些程序中，你会看到已经废弃的*pragma*，如*use attrs*，它的功能



现在已经由子例程声明语法直接支持，或者可以看到`use vars`，我们已经把它替换为`our`声明。

我们并不急于打破原来的方式，不过我们确实认为新方式更好一些。

在这一章的最后，我们会展示如何创建你自己的`pragma`，就像是Perl本身提供的`pragma`一样。

## attributes

```
sub afunc : method;
my $closure = sub : method { ... };

use attributes;
@attrlist = attributes::get(\&afunc);
```

`attributes` pragma有两个用途。第一个作用是提供一种声明属性列表（attribute lists）的内部机制，这些是与子例程声明和（很快就会有的）变量声明关联的可选属性（由于这是一个内部机制，通常你不会直接使用这个`pragma`）。第二个作用是提供一种方法，可以使用`attributes::get`函数在运行时获取这些属性列表。如果要实现这个作用，`attributes`只是一个标准模块，而不是`pragma`。

目前Perl只处理了很少的几个内置属性。包特定的属性由一种试验性扩展机制支持，这种机制在`attributes(3)`手册页的“包特定属性处理”一节中介绍。

属性设置在编译时发生；如果试图设置一个无法识别的属性，这将是一个编译错误（这个错误可以由`eval`捕获；不过还是会在该`eval`块中停止编译）。

对于子例程，目前只实现了3个内置属性：`locked`、`method`和`lvalue`。另外参见第7章了解这些属性的更多介绍。对于变量，目前则没有像子例程那样的内置属性，不过我们可以考虑几个可能喜欢的属性，如`constant`。

`attributes` pragma提供了两个通用的子例程。如果做出请求可以导入这两个子例程。

### get

给定一个输入参数，而且这个参数是一个子例程或变量的引用，那么这个函数会返回一个（可能为空的）属性列表。如果传入非法的参数，这个函数会通过调用`Carp::croak`产生一个异常。

### reftype

这个函数的工作有点类似于内置`ref`函数，不过它总是返回所引用值的底层内置Perl数据类型，而忽略它的祝福包。

属性处理的具体细节还不是很明确，所以最好查看你的Perl版本提供的在线文档来了解最新进展。

## autodie

```
use autodie;
```

这个pragma将Perl函数调用的失败转换为致命错误，不过仅限于其词法作用域中。一些标准Perl函数失败时会返回false，这个pragma则将这些Perl函数替换为另一个函数版本，即失败时会抛出异常。基于\$@中的异常消息，你可以知道遇到的是哪种异常：

```
eval {
    use autodie;
    open my $fh, "<:encoding(UTF-8)", $filename;
    my @lines = <$fh>;
    close $fh;
}

for ($@) {
    when (undef) { }
    when ("open") { say "Open failed"; }
    when (":io") { say "Some other IO error"; }
    when (":all") { say "Some other autodie error" }
    default { say "Non-autodie error" }
}
```

这个pragma还可以替换相关的函数组，如以上处理输入和输出的函数：

```
use autodie qw(:io);
```

如果你不希望一个内部作用域启用这个特性，可以将它关闭：

```
no autodie;
```

## autouse

```
use autouse "Carp" => qw(carp croak);
carp "this carp was predeclared and autoused";
```

这个pragma提供了一种运行时请求加载特定模块的机制，前提是已经调用了该模块的一个函数。为实现这一点，它提供了一个桩函数，一旦触发会将它自身替换为真正的函数调用。这有些类似于标准AutoLoader和SelfLoader模块的工作。简单地讲，这是一个提高性能的技巧，有些模块在实际执行中永远不会被调用，通过避免编译这些模块，这有助于你的Perl程序更快地启动（一般来讲是这样）。

autouse的表现如何取决于模块是否已经加载。例如，如果模块Module已经加载，以下声明：

```
use autouse "Module" => qw(func1 func2($;$) Module::func3);
```

等价于以下简单地导入两个函数：

```
use MODULE qw(func1 func2);
```

这里假设这个模块定义func2有原型 (\$;\$)，另外func1和func3没有原型（更一般地，这里还假设这个模块使用Exporter的标准import方法；否则，会产生一个致命错误）。总之，它会完全忽略Module::func3，因为原先是这样声明的。

不过，如果解析autouse pragma时这个模块尚未加载，这个pragma会声明函数func1和func2在当前包中。它还会声明一个函数Module::func3（这有点反社会，当然如果Module模块不存在，后果更严重）。调用这些函数时，它们会确保当前模块已经加载，然后将自身调用替换为刚加载的实际函数。

因为autouse pragma将程序中某些部分的执行从编译时转移到运行时，这可能会有让人不快的衍生产物。例如，你要autouse的模块有一些初始化工作，希望较早完成，而由于autouse pragma，这些工作可能无法足够早地完成。这种autouse还可能隐藏代码中的bug，因为一些重要的检查从编译时移到了运行时。

具体地，如果你在autouse行上指定的原型是错误的，但只有到你真正执行相应的函数时才有可能发现这一点，对于一个很少调用的函数，这可能是几个月甚至几年之后。为了部分缓解这个问题，可以在代码开发中如下编写代码：

```
use Chase;
use autouse Chase => qw[ hue($) cry(&$) ];
cry "this cry was predeclared and autoused";
```

第一行确保参数规范中的错误会尽早发现。程序从开发阶段进入生产阶段时，可以注释掉正常的Chase模块加载，只留下autouse调用。这样一来，既能在开发阶段得到安全性，又能在生产阶段保证性能。

## base

```
use base qw(Mother Father);
```

这个pragma允许程序员基于所列的父类很方便地声明一个派生类。这个pragma基本上已经过时，大多数人更愿意使用parent pragma。

上面的声明大致等价于：

```
BEGIN {
    require Mother;
    require Father;
    push @ISA, qw(Mother Father);
```



```
}
```

`base pragma`负责所需的所有`require`。`strict "vars"pragma`起作用时，`use base`（实际上）允许你为`@ISA`赋值而无需首先声明`our @ISA`（由于`base pragma`在编译时发生，最好避免在运行时自行调整`@ISA`）。

不过，除此以外，`base`还有另外一个性质。如果任何命名基类使用了`fields`下的字段功能（本章后面将会提到），这个`pragma`会由这个基类初始化包的特殊字段属性（不支持类的多重继承。如果多个命名基类都有字段，`base pragma`会产生一个异常）。

任何尚未加载的基类都会通过`require`自动加载。不过，是否用`require`加载一个基类包并不是通过通常的`%INC`审查来决定，而是由包中有没有一个全局的`$VERSION`来确定。这个技巧可以保证Perl不会反复尝试加载一个不在其可加载文件中的基类（比如要作为另外某个模块文件的一部分加载），否则这种尝试最终也会失败。如果成功加载一个文件之后未能检测到`$VERSION`，`base`会在基类包中定义`$VERSION`，将它设置为字符串`"-1, defined by base.pm"`。这个`pragma`的以后版本中这个字符串可能还会改变。

## bigint

```
use bigint;
```

这个`pragma`绕过依赖于体系结构的整数操作处理，来处理非常大的数以及特殊值`NaN`（这表示不是一个数）：

```
use bigint;
say 2 ** 512;
```

这个`pragma`通过重载数值操作符使用`Math::BigInt`来计算值。因此，这可能比内置的操作慢得多。不过，尽管慢但正确的答案总比快但是错误的答案要好。

可以加载不同的实现库，它们在性能上可能不同。默认地，`bigint`会使用一个纯Perl实现，不过如果有更快的库也可以加载那个更快的库：

```
use bigint lib => 'GMP';
```

可以设置精确性，这会确定答案中的有效数字个数：

```
use bigint a => 2;
```

或者，可以设置精度，这会指定答案的量级。精度小于0会被忽略：

```
use bigint p => -2; # 精确到百分位（忽略）
use bigint p => 1;  # 取整为10
```

## bignum

```
use bignum;
```

这个pragma绕过依赖于体系结构的数值操作处理，来处理非常大的数（或者有更多小数位的数），以及处理特殊值NaN（这表示不是一个数）：

```
use bignum;
say sqrt(2);
```

这个pragma通过重载数值操作符使用Math::BigInt和Math::BigFloat来计算值。参见bigint。

## bigrat

```
use bigrat;
```

这个pragma绕过依赖于体系结构的数值操作处理，来处理有理数（也就是说，分数），并保持它们为有理数，从而不会损失精度（至少在你做好准备之前不会损失精度）：

```
use bigrat;
say 1/2 + 1/3; # 5/6
```

这个pragma通过重载数值操作符使用Math::BigInt和Math::BigRat来计算值。参见bigint。

## blib

从命令行：

```
% Perl-Mblib program [args...]
% Perl-Mblib=DIR program [args...]
```

从Perl程序：

```
use blib;
use blib DIR;
```

这个pragma主要用于使用Perl的-M命令行开关对一个卸载的包版本测试任意的Perl程序。它假设你的目录结构由标准ExtUtils::MakeMaker或Module::Build模块生成。

这个pragma首先从名为DIR的目录（如果没有指定，则为当前目录）开始查找blib目录，如果没有找到blib目录，再回到..目录，最多能向上扫描5级父目录。

## bytes

```
use bytes;  
no bytes;
```

`bytes pragma`禁用它所在的词法作用域中其余部分的字符语义。`no bytes pragma`可以用于在当前词法作用域中实现`use bytes`相反的效果。

Perl通常假设有字符数据时就使用字符语义（也就是说，数据源标志为采用某种特定的字符编码）。

要了解字符语义和字节语义之间的影响和差别，参见第6章。去趟东京可能也会有帮助。

你可能并不想使用这个`pragma`，而且在将来的Perl版本中它很可能会消失。在v5.14中，`bytes pragma`的文档强烈建议不要使用这个`pragma`。如果你有一个字符串，就应该把它当作字符串，而不用担心它的底层编码。

## charnames

```
use charnames HOW;  
print "\N{CHARNAME} is a funny character";  
  
use charnames ();    # 没有编译时\N{}, 只有运行时函数
```

除了`use charnames ()`，所有形式都使用`\N{CHARNAME}`记法，允许将命名字符内插到字符串和正则表达式：

```
use charnames ":full";  
print "\N{GREEK SMALL LETTER SIGMA} is called sigma.\n";  
  
use charnames ":short";  
print "\N{greek:Sigma} is a uppercase sigma.\n";  
  
use charnames qw(cyrillic greek);  
print "\N{sigma} is Greek sigma, and \N{be} is Cyrillic b.\n";  
  
use charnames ":full", ":alias" => {  
    "WRY CAT" => "CAT FACE WITH WRY SMILE",  
    "AMELIA" => "DROMEDARY CAMEL",  
    "s with comma" => 0x0219,  
};  
  
# ":loose" 只在v5.16及以后版本中支持  
use charnames ":loose";
```

如果有`:full`，会扩展`\N{CHARNAME}`，首先查看标准Unicode字符名列表中的第一个字符名。如果有`:short`，而且`CHARNAME`形式为`SCRIPT:CNAME`，则在文字系统`SCRIPT`中查找`CNAME`（作为一个字母）。如果有`:loose`（而且你运行的是v5.16或更高版本），它的工作类似于`:full`，不过查找名不考虑大小写、空白符或下划线，就像正则表达式中的Unicode属性名。



结合一个或多个Unicode文字系统名参数使用时<sup>注1</sup>，会在给定的文字系统中查找CHARNAME（作为一个字母），首先查看第一个列出的文字系统，然后是第二个（如果有），如此继续。要查找一个给定文字系统SCRIPTNAME中的CHARNAME，会查找以下名字：

```
SCRIPTNAME CAPITAL LETTER CHARNAME
SCRIPTNAME SMALL LETTER CHARNAME
SCRIPTNAME LETTER CHARNAME
```

如果CHARNAME是全小写，则忽略CAPITAL变体。否则，忽略SMALL变体。

```
use charnames "Greek";
print "\N{Sigma} \N{sigma} \N{final sigma}\n"; # Σ σ ς
use charnames "Latin";
print "\N{DZ} \N{D with small letter z} \N{dz}\n"; # DZ Dz dz
```

\N{NAME}只在编译时作为双引号字符串中使用的一个特殊形式的字符串常量进行处理。NAME必须是一个直接量，不能在\N{NAME}中使用变量。对于类似的运行时功能，要使用charnames::string\_vianame，如下所述。

记法\N{U+HEXDIGITS}（这里HEXDIGITS是一个十六进制数）也会把一个Unicode字符插入到一个字符串中，不过与所有\N{...}用法不同，这个记法不需要charnames pragma。插入的字符就是码点（序数值）等于这个十六进制数的字符。例如，\N{U+263A}是Unicode的（白色背景、黑色前景的）笑脸。这个记法不需要这个pragma，使用字符名的等价方法\N{WHITE SMILING FACE}则需要charnames pragma。

包含一个\N{CHARNAME}或\N{U+HEXDIGITS}的字符串会自动具有Unicode语义，即使你没有使用Unicode字符串特性。

C0和C1控制字符（U+0000..U+001F, U+0080..U+009F）没有正式的Unicode名，不过可以使用ISO 6429名：LINE、FEED、ESCAPE及其缩写LF、ESC等。另外参见charnames手册页来了解其他常用的别名。

如果输入名未知，\N{NAME}会产生一个警告，并替换Unicode REPLACEMENT字符（U+FFFD）。

对于\N{NAME}，如果bytes起作用，而且输入名是一个无法用字节表示的字符名（也就是说，序数值大于255），则是一个致命错误。

## 定制字符名

可以创建定制字符名，从而可以更方便地键入字符，或者为Unicode尚未指定名字的码点指定名字。可以使用匿名散列增加别名：

---

注1：可以在手册页中找到当前Unicode可识别的文字系统列表。

```
use charnames ":alias" => {
    e_ACUTE => "LATIN SMALL LETTER E WITH ACUTE",
    "APPLE LOGO" => 0xF8FF, # 私用的码点
};
my $str = "\N{APPLE LOGO}";
```

或者使用一个包含键/值对列表的文件来增加别名：

```
use charnames ":alias" => "pro"; # 查找unicore/pro_alias.pl
```

指定的文件应当在@INC路径中的某个`unicore/`子目录下，而且文件名末尾应当有一个`_alias.pl`。所以，举例来说，以上查找的文件将是`unicore/pro_alias.pl`。这个文件应当返回一个纯Perl列表：

```
(
    A_grave => "LATIN CAPITAL LETTER A WITH GRAVE",
    A_circ  => "LATIN CAPITAL LETTER A WITH CIRCUMFLEX",
    A_diaer => "LATIN CAPITAL LETTER A WITH DIAERESIS",
    A_dier  => "LATIN CAPITAL LETTER A WITH DIAERESIS",
    A_uml   => "LATIN CAPITAL LETTER A WITH DIAERESIS",
    A_tilde => "LATIN CAPITAL LETTER A WITH TILDE",
    A_macron => "LATIN CAPITAL LETTER A WITH MACRON",
);
```

这些方法都会自动插入`:full`作为第一个参数（如果没有给定任何其他参数），也可以显式指定`:full`：

```
use charnames ":full", ":alias" => "pro";
```

甚至私用的字符也可以得到名字，例如，如果有以下代码：

```
use charnames ":full", ":alias" => {
    "TENGWAR LETTER TINCO"    => 0xE000,
    "TENGWAR LETTER PARMA"    => 0xE001,
    "TENGWAR LETTER CALMA"    => 0xE002,
    "TENGWAR LETTER QUESSE"   => 0xE003,
    "TENGWAR LETTER ANDO"     => 0xE004,
    ...
}
```

这个词法作用域中的字符串和正则表达式可以引用这些命名字符：

```
if (/\\N{TENGWAR LETTER TINCO}/) { ... }
```

## 运行时查找

这个`pragma`还提供了3个函数来完成运行时字符名和字符码点之间的转换，而不是像`\N{CHARNAME}`内插那样在编译时完成。这包括：

`charnames::vianame`

取一个正式名、正式别名或定制别名，返回一个整数码点。例如，它将字符串“LATIN

SMALL LETTER A"转换为0x61。

`charnames::string_vianame`

取一个可能是正式名、正式别名或命名序列的字符串，返回一个字符串。例如，它将字符串"LATIN SMALL LETTER A"转换为"a"。返回的字符串可能长度大于1（不过这种情况很少）。

`charnames::viacode`

取一个整数，如果有相应的正式别名则返回一个正式别名，否则返回正式名。例如，这会把0x61转换为字符串"LATIN SMALL LETTER A"。只有不存在正式名时才会返回定制名，如私用的码点。

这些函数没有导出，所以使用时必须完全限定。它们还允许在运行时访问你可能创建的所有定制别名。下面的代码展示了这些函数是如何工作的：

```
use v5.14;
use warnings;
use warnings FATAL => "utf8";
use open qw(:std :utf8);

use charnames ":full", ":alias" => {
    ecute => "LATIN SMALL LETTER E WITH ACUTE",
    "APPLE LOGO" => 0xF8FF, # 私用字符
};

printf "U+%04X is named '%s'.\n", 0xE9 => charnames::viacode(0xE9);
printf "%s is code U+%04X.\n", ecute => charnames::vianame("ecute");
printf "%s is string '%s'.\n", ecute => charnames::string_vianame("ecute");

printf "U+%04X is named '%s'.\n", 0xF8FF => charnames::viacode(0xF8FF);
printf "%s is code U+%04X.\n", "APPLE LOGO" => charnames::vianame("APPLE LOGO");
printf "%s is string '%s'.\n", "APPLE LOGO" => charnames::string_vianame("APPLE LOGO");
```

所生成的输出如下：

```
U+00E9 is named 'LATIN SMALL LETTER E WITH ACUTE'.
ecute is code U+00E9.
ecute is string 'é'.

U+F8FF is named 'APPLE LOGO'.
APPLE LOGO is code U+F8FF.
APPLE LOGO is string '🍏'.
```

甚至可以写你自己的模块完成类似`charnames` pragma的工作，但是以不同方式定义字符名。不过，这个接口还是试验性的，所以参见手册页来了解最新进展。

## constant

```
use constant BUFFER_SIZE => 4096;
use constant ONE_YEAR => 365.2425 * 24 * 60 * 60;
```



```

use constant PI          => 4 * atan2 1, 1;
use constant DEBUGGING   => 0;
use constant ORACLE       => 'oracle@cs.indiana.edu';
use constant USERNAME     => scalar getpwuid($<);
use constant USERINFO     => getpwuid($<);

use constant {
    BUFFER_SIZE          => 4096,
    ONE_YEAR              => 365.2425 * 24 * 60 * 60,
    PI                    => 4 * atan2( 1, 1 ),
    DEBUGGING             => 0,
    ORACLE                 => 'oracle@cs.indiana.edu',
    USERNAME              => scalar getpwuid($<),
    USERINFO              => getpwuid($<),
};
sub deg2rad { PI * $_[0] / 180 }

print "This line does nothing" unless DEBUGGING;

# 引用可以声明为常量
use constant CHASH        => { foo => 42 };
use constant CARRAY       => [ 1,2,3,4 ];
use constant CCODE        => sub { "bite $_[0]\n" };

print CHASH->{foo};
print CARRAY->[$i];
print CCODE->("me");
print CHASH->[10];          # 编译时错误

```

这个pragma声明命名符号是一个有给定标量或列表值的不可变的常量<sup>注2</sup>。值在列表上下文中计算。可以用scalar覆盖（如上所示）。给定一个散列引用，只需要一个use语句就可以一次声明多个常量。

由于这些常量前面没有一个\$，不能将它们直接内插到双引号字符串中，不过可以间接实现内插：

```
print "The value of PI is @{{ PI }}.\n";
```

由于列表常量作为列表返回，而不是作为数组，必须另外加一对小括号为列表值常量加下标，就像所有其他列表表达式一样：

```

$homedir = USERINFO[7];      #不正确
$homedir = (USERINFO)[7];    #正确

```

尽管通常建议对常量使用全大写字母（另外单词之间使用下划线），使它们更突出，这也有助于避免与其他关键字和子例程名可能的冲突，但这只是一个约定。常量名必须以一个大写字母或一个下划线开头，不过（如果是字母）并不要求它是一个大写或标题形式的字母。

---

注2： 实现为不带参数的子例程，每次返回相同的常量。

常量并不是它所出现的词法作用域所私有。实际上，对于做出这个声明的包来说，常量只是这个包的符号表中的无参数子例程。包Other中的常量CONST可以用Other::CONST表示。有关这种子例程的编译时内联的更多内容，请参见第7章中“内联常量函数”一节。

与所有use指令类似，use constant在编译时发生。因此，最容易引起误解的是把常量声明放在一个条件语句中，如if (\$foo) { use constant ... }。由于这在编译时发生，Perl在遇到这个常量声明时会把常量表达式替换为它们的值。

如果声明中省略了符号的值，在标量上下文中则会为它指定值为undef，在列表上下文中会给定空列表()。不过最好还是显式地做出声明：

```
use constant CAMELIDS => ();
use constant CAMEL_HOME => undef;
```

## 关于常量的限制

目前列表常量的内联方式与标量常量的内联方式不同。而且不能有与常量同名的子例程或关键字。这可能是件好事。

不能作为列表一次声明多个命名常量：

```
use constant FOO => 4, BAR => 5;      #不正确
```

这会定义一个名为FOO的常量，它的返回列表是(4, "BAR", 5)。要想声明多个常量，需要这样做：

```
use constant FOO => 4
use constant BAR => 5;
```

或者甚至这样做：

```
use constant {
    FOO => 4,
    BAR => 5,
};
```

如果你在一个会自动对裸名加引号的上下文中使用常量，那你是自找麻烦（这一点同样适用于所有子例程调用，而不只是常量）。例如，不能使用\$hash{CONSTANT}，因为CONSTANT会被解释为一个字符串。要用\$hash{CONSTANT()}或\$hash{+CONSTANT}以避免引号机制介入。类似地，由于=>操作符会对其左操作数加引号（如果左操作数是一个裸名），所以必须写为CONSTANT() => "value"而不是CONSTANT=> "value"。

## deprecate

```
use deprecate;
```

已经标志为从标准库删除的内核模块会使用这个pragma发出一个警告，告诉你应当使用CPAN版本。如果你的模块不是标准库中的模块，这个pragma什么也不做。

## diagnostics

```
use diagnostics;           # 编译时启用
use diagnostics -verbose;

enable diagnostics;        # 运行时启用
disable diagnostics;       # 运行时禁用
```

这个pragma扩展普通的简洁诊断，并抑制重复的警告。它会补充简短的版本，加入*perldiag*中的解释和描述。与其他pragma类似，它也会影响程序的编译阶段，而不只是运行阶段。

在程序最前面声明`use diagnostics`时，这会自动启用Perl的`-w`命令行开关，将`$^W`设置为1。整个编译的其余阶段都会处于这种增强的诊断状态。警告仍输出到STDERR。

由于运行时和编译时问题之间的交互，不能使用`no diagnostics`在编译时将其关闭（而且这可能也不是一个好的想法）。不过，可以使用`disable`和`enable`方法控制其在运行时的行为（要确保首先`use`，否则无法使用这些方法）。

`-verbose`标记首先打印出*perldiag*手册页的介绍，然后才会给出其他诊断。可以（在`use`之前）设置`$diagnostics::PRETTY`变量，从而为分页工具（如`less(1)`或`more(1)`）生成更漂亮的转义序列：

```
BEGIN { $diagnostics::PRETTY = 1 }
use diagnostics;
```

从Perl发出并由这个pragma检测的各个警告只显示一次。如果你所在的循环会反复地生成相同的警告（如未初始化值），这个pragma就很有用。手动生成的警告（如由`warn`或`carp`调用生成的警告）不受这种重复检测机制的影响。

下面给出使用`diagnostics pragma`的一些例子。下面的文件会在运行时和编译时触发一些错误：

```
use diagnostics;
print NOWHERE "nothing\n";
print STDERR "\n\tThis message should be unadorned.\n";
warn "\tThis is a user warning";
print "\nDIAGNOSTIC TESTER: Please enter a<CR> here: ";
my $a, $b = scalar <STDIN>;
print "\n";
print $x/$y;
```

输出如下：



Parentheses missing around "my" list at diag.pl line 6 (#1)

(W parenthesis) You said something like

```
my $foo, $bar = @_;
```

when you meant

```
my ($foo, $bar) = @_;
```

Remember that "my", "our", "local" and "state" bind tighter than comma.

Name "main::y" used only once: possible typo at diag.pl line 8 (#2)

(W once) Typographical errors often show up as unique variable names. If you had a good reason for having a unique name, then just mention it again somehow to suppress the message. The our declaration is provided for this purpose.

NOTE: This warning detects symbols that have been used only once so \$c, @c, %c, \*c, &c, sub c{}, c(), and c (the filehandle or format) are considered the same; if a program uses \$c only once but also uses any of the others it will not trigger this warning.

Name "main::b" used only once: possible typo at diag.pl line 6 (#2)

Name "main::NOWHERE" used only once: possible typo at diag.pl line 2 (#2)

Name "main::x" used only once: possible typo at diag.pl line 8 (#2)

print() on unopened filehandle NOWHERE at diag.pl line 2 (#3)

(W unopened) An I/O operation was attempted on a filehandle that was never initialized. You need to do an open(), a sysopen(), or a socket() call, or call a constructor from the FileHandle package.

This message should be unadorned.

This is a user warning at diag.pl line 4.

DIAGNOSTIC TESTER: Please enter a<CR> here:

Use of uninitialized value \$y in division (/) at diag.pl line 8, <STDIN> line 1 (#4) (W uninitialized) An undefined value was used as if it were already defined. It was interpreted as a "" or a 0, but maybe it was a mistake. To suppress this warning assign a defined value to your variables.

To help you figure out what was undefined, Perl will try to tell you the name of the variable (if any) that was undefined. In some cases it cannot do this, so it also tells you what operation you used the undefined value in. Note, however, that Perl optimizes your program and the operation displayed in the warning may not necessarily appear literally in your program. For example, "that \$foo" is usually optimized into "that ". \$foo, and the warning will refer to the concatenation (.) operator, even though there is no . in your program.

Use of uninitialized value \$x in division (/) at diag.pl line 8, <STDIN> line 1 (#4)

Illegal division by zero at diag.pl line 8, <STDIN> line 1 (#5)

(F) You tried to divide a number by 0. Either something was wrong in your logic, or you need to put a conditional in to guard against

```
meaningless input.
```

```
Uncaught exception from user code:
```

```
Illegal division by zero at diag.pl line 8, <STDIN> line 1.  
at diag.pl line 8
```

诊断消息来自`perldiag`手册页。如果发现已经有一个`$SIG{__WARN__}`处理器，仍会接受这个处理器的处理，不过要在`diagnostics::splainthis`函数（`pragma`的`$SIG{__WARN__}`拦截器）已经处理完警告之后。Perl目前并不支持堆叠的处理器，所以这是我们目前最好的做法了。如果你对哪些东西会被拦截实在好奇，可以设置一个`$diagnostics::DEBUG`变量：

```
BEGIN { $diagnostics::DEBUG = 1 }  
use diagnostics;
```

## encoding

```
use encoding ENCODING;  
use encoding "euc-jp";
```

这个`pragma`允许你用你喜欢的任何`ENCODING`写Perl代码，由Perl正确地转换字符串，并把标准输出和错误转换为指定的编码。不过，它从来没有正确工作过，可能永远也不会。另一种做法是，可以把源代码从你正在使用的遗留编码转换为UTF-8，再在文件最上面加一个`use utf8`声明。设置标准I/O流使用`open pragma`或者使用`binmode`。

## feature

```
use feature ":5.10"; # 这是一个"特性包"  
use feature qw(say state switch unicode_strings);  
  
{  
    no feature qw(say);  
    ...;  
}
```

Perl通过`feature pragma`来引入新的关键字和特性，保证自己不会过时。这个`pragma`可以在词法作用域中启用或禁用特性。通过一个版本标志或特性名来指定要打开或关闭的特性。

**say**

启用`say`关键字，这就类似于`print`，但是有一个换行符。

**state**

启用`state`，这支持持久存储的子例程变量。

**switch**

启用Perl的增强型`switch`结构，名为`given-when`。

## unicode\_strings

这个特性不是一个关键字。实际上，它会导致词法作用域中的所有字符串操作使用Unicode语义。这个特性还应用于这个作用域中编译的正则表达式（即使它们最终会在这个作用域之外执行）。另外参见*perlunicode*中的“Unicode Bug”。这是唯一不在:v5.10中的一个特性，不过:v5.12及以后版本都有这个特性。

## fields

这个pragma在v5.10中已经废弃，所以我们不打算对它做过多介绍，这也表示不建议你使用它。这个pragma设计为可以用来声明类字段，从而可以在编译时进行类型检查。为此，它依赖于（已经删除的）伪散列特性。如果你还无法摆脱遗留代码中的字段，可能还会在相应的文档中看到它，（至少）v5.14中还有这个pragma。

## filetest

```
$scan_perhaps_read = -r "file";      # 使用模式位
{
    use filetest "access";            # 凭直觉知道更难
    $scan_really_read = -r "file";
}
$scan_perhaps_read = -r "file";      # 再次使用模式位
```

这是一个词法作用域pragma，告诉编译器改变一元文件测试符-r，-w，-x，-R，-W和-X的行为，见第3章的介绍。这些文件测试的默认行为是使用stat系列调用返回的模式位。不过，这可能并不一定正确，如当文件系统了解ACL（访问控制列表）时。在类似AFS等重视权限的环境中，filetest pragma可以帮助权限操作符返回与其他工具更一致的结果。

使用filetest时，受影响的文件测试可能会有一些性能损失，因为有些系统上需要模拟这些扩展的功能。

警告：如果出于安全考虑而没有使用文件测试，这个想法从一开始就是错误的。这会为竞态条件打开一个窗口，因为没有办法保证在测试和实际操作之间权限不会改变。如果你对安全考虑很少，就不要使用文件测试操作符来确定某个事情是否可行。实际上，应该直接尝试这个实际操作，然后测试操作是否成功。（必须如此）。另外参见第20章的“处理计时问题”一节。

## if

```
use if CONDITION, MODULE => IMPORTS;

use if $^O =~ /MSWin/, "Win32::File";
```



```
use if $^V >= 5.010, parent => qw(Mojolicious::UserAgent);
use if $^V < 5.010, base => qw(LWP::UserAgent);
```

`if pragma`根据某个条件控制一个模块的加载。这个`pragma`并不处理指定最小版本的模块加载。需要在模块名后面指定一个导入列表。

## inc::latest

```
use inc::latest "Module::Build";
```

一些模块作者开始将其发行版本中的依赖包分布在`inc`目录中。例如，他们想使用某个特定版本的`Module::Build`，所以会在其发行版本中把这个模块安装在`inc`，并让它优先于已安装的版本。在Perl工具链了解`configure_requires`之前，可以利用这个技巧在发行版本开始模块的构建过程。

`inc::latest`模块告诉perl加载`inc`中的一个版本，前提是这个版本要大于其余`@INC`中安装的版本。

## integer

```
use integer;
$x = 10/3;
# $x 现在是3，而不是3.3333333333333333
```

这是一个词法作用域`pragma`，告诉编译器从这里开始直到外围块的末尾都使用整数操作。在很多机器上，这对于大多数计算并没有太大影响，但是对于其余没有浮点数硬件的体系结构来说（很少），这会带来显著的性能差异。

需要说明，这个`pragma`会影响某些数值操作，而不是数字本身。

例如，如果运行以下代码：

```
use integer;
$x = 1.8;
$y = $x + 1;
$z = -1.8;
```

你会得到`$x == 1.8`，`$y == 2`和`$z == -1`。`$z`之所以得到这个结果，是因为一元`-`会作为一个操作，所以值1.8在其符号位翻转之前截断为1。类似地，接收浮点数的函数（如`sqrt`或三角函数）即使在`use integer`下也会接收和返回浮点数。所以`sqrt(1.44)`是1.2，而`0 + sqrt(1.44)`现在只是1。

这里使用了C编译器提供的原生整数运算。这说明Perl自己的算术操作语义可能不会保留。可能导致麻烦的一个常见问题是负数的取模。Perl可能会以一种方式处理，而你的硬件可能会以另一种方式处理：

```
% Perl-le "print (4 % -3)"
-2

% Perl-Minteger -le "print (4 % -3)"
1
```

另外，整数运算会使位操作符将其操作数处理为有符号值，而不是无符号值：

```
% Perl-le "print ~0"
18446744073709551615

% Perl-Minteger -le "print ~0"
-1
```

## less

```
use less;

use less "CPU";
use less "memory";
use less "time";
use less "disk";
use less "fat";          # great with "use locale";
```

这个pragma在v5.10及以后版本中实现，其目的是为编译器、代码生成器或解释器提供提示，通过使用caller现在返回的新的提示散列引用来支持某种权衡。

这个模块一直是Perl发行版本的一部分（作为一个玩笑），不过在v5.10之前它什么都不做。甚至在v5.10版本中，提示只在其词法作用域中可用，所以尽管按pragma文档的说法，好像另一个模块可以很容易地找出你希望哪些特性“少一点”，但这仍然只是新的caller特性的一个演示。

如果要求少使用一些特性，而Perl目前还不知道如何减少使用，这不算是一个错误。

## lib

```
use lib "$ENV{HOME}/libperl"; # 增加 ~/libperl
no lib ".";                  # 删除cwd
```

这个pragma简化了编译时@INC的管理。它通常用于向Perl的搜索路径增加额外的目录，使得以后do、require和use语句无法在Perl默认搜索路径中找到库文件时可以在这些目录中查找。这对于use尤其重要，因为它也在编译时发生，而且如果（在运行时）正常地设置@INC就为时过晚了。

use lib的参数会追加到Perl搜索路径的前面。use lib LIST几乎等同于BEGIN { unshift(@INC, LIST) }，不过use lib LIST还支持平台特定的目录。对于其参数列表中每一个给定的目录\$dir，lib pragma还会查看是否存在一个名为\$dir/\$archname/auto的目

录。如果是这样，则认为`$dir/$archname`目录是一个相应的平台特定的目录，所以它会增加到`@INC`（在`$dir`的前面）。

冗余地增加目录会减慢访问速度，并浪费少量内存，为了避免这种情况，增加目录时会从`@INC`删除末尾重复的目录。

正常情况下，应当只向`@INC`中增加目录。如果你确实需要从`@INC`删除目录，要注意只删除你自己增加的那些目录，或者可以肯定程序中其他模块不会用到的那些目录。其他模块可能也向你的`@INC`增加了目录以保证其正确的操作。

`no lib pragma`从`@INC`删除指定的各个目录的所有实例。它还会如前所述删除所有相应的平台特定的目录。

加载`lib pragma`时，它会把`@INC`的当前值保存到数组`@lib::ORIG_INC`中。所以要恢复原来的值，只需要把那个数组复制到真正的`@INC`。

尽管`@INC`通常包含点号`(.)`即当前目录，但这并没有你想象中那么有用。一方面，点记录出现在最后，而不是一开始，所以当前目录中安装的模块不会突然覆盖系统版本的模块。如果你确实希望这么做，也可以声明`use lib "."`。更烦人的是，它是Perl进程的当前目录，而不是安装这个脚本的目录，所以并不可靠。如果你创建了一个程序以及这个程序要使用的一些模块，在开发阶段它能够正常工作，但是如果不在这些文件所在的目录中运行，程序就不能正常工作了。

对此的一个解决方案是使用标准FindBin模块：

```
use FindBin;           # 脚本安装在哪里？
use lib $FindBin::Bin; # 还要使用该目录来提供库
```

FindBin模块尝试猜出当前运行的程序所安装目录的完整路径。出于安全性的考虑，不要使用这个路径，因为恶意程序如果足够努力是可以欺骗它的。不过，除非你有意破坏这个模块，否则它应该能正常工作。这个模块提供了一个`$FindBin::Bin`变量（你可以导入），其中包含这个模块猜测的程序的安装路径。可以再使用`lib pragma`把这个目录增加到你的`@INC`，这会生成一个相对于可执行程序的路径。

有些程序希望安装在`bin`目录，然后在与`bin`同级的一个`lib`目录中安装的“同辈”文件中查找其库模块。例如，程序可能放在`/usr/local/apache/bin`或`/opt/perl/bin`，而库在`/usr/local/apache/lib`和`/opt/perl/lib`中。以下代码可以很好地处理这一点：

```
use FindBin qw($Bin);
use lib "$Bin/../lib";
```

如果发现你在多个无关的程序中指定相同的`use lib`，可以考虑设置`PERL5LIB`环境变量。另外参见第17章中有关`PERL5LIB`环境变量的介绍。



```
# sh, bash, ksh或zsh的语法
$ PERL5LIB=$HOME/perl5lib; export PERL5LIB

# csh或tcsh的语法
% setenv PERL5LIB ~/perl5lib
```

如果你想要在这个程序上使用可选的目录，但不改变其源代码，可以使用`-I`命令行开关：

```
% Perl -I ~/perl5lib program-path args
```

参见第17章了解更多有关在命令行上使用`-I`开关的信息。

## locale

```
@x = sort @y;          # ASCII排序顺序
{
    use locale;
    @x = sort @y;      # 本地化环境定义的排序顺序
}
@x = sort @y;          # 恢复为ASCII排序顺序
```

这是一个词法作用域pragma，告诉编译器对内置操作启用（在`no locale`下则是禁用）POSIX本地化环境。启用本地化环境会告诉Perl实现字符串比较和区分大小写的功能时要考虑你的POSIX语言环境。如果这个pragma起作用，而且你的C库了解POSIX本地化环境，对于正则表达式，Perl会查看你的`LC_CTYPE`设置，对于字符串比较（如`sort`中的比较），则会查看你的`LC_COLLATE`设置。

由于本地化环境更应算是一种本地化而不是国际化，所以使用本地化环境可能与Unicode之间有奇怪的交互。更可移植而且更可靠的做法是使用Perl的原生Unicode功能来完成大小写转换和比较，这在所有安装上都是标准的，而不依赖于某些可能很奇怪的开发商本地化环境。另外参见第6章中“Unicode文本比较与排序”和“本地化排序”小节。

## mro

```
use mro;                # 全局启用next::method及类似方法

use mro "dfs";          # 对这个类启用DFS MRO (Perl默认)
use mro "c3";           # 对这个类启用C3 MRO
```

默认地，Perl会在`@INC`中对类（包名）使用一个深度优先搜索算法（DFS）来搜索方法。`mro` pragma会改变方法解析顺序。指定`dfs`会使用默认的深度优先搜索，而指定`c3`将使用C3算法来解决多重继承中的某些二义性。如果没有一个导入列表，可以启用与C3方法解析交互的特性，来保持默认的方法解析顺序（参见第12章）。

# open

```
use open IN => ":crlf", OUT => ":raw";
use open OUT => ":utf8";
use open IO => ":encoding(iso-8859-7)";

use open IO => ":locale";

use open ":encoding(utf8)";
use open ":locale";
use open ":encoding(iso-8859-7)";

use open ":std";
```

`open pragma`为I/O操作声明一个或多个默认层（原来称为原则*discipline*），不过前提是你的Perl库安装有PerlIO。对于这个pragma的词法作用域中找到的所有`open`和`readpipe`（也就是`qx//`或反引号）操作符，如果没有指定它们自己的层，就会使用所声明的默认层。如果`open`有显式设置的层，则不会受这个pragma的影响，另外，不论什么情况下，`sysopen`都不会受此影响。

有多个层可以选择：

## :bytes

这个层把数据处理为码点在0到255之间的字符。这与`:utf8`层正好相反。不过，它不同于`:raw`，因为在Windows系统下它还会是完成CRLF处理。

## :crlf

这一层对应于文本模式，其中行结束符会与原生行结束符来回转换。如果在一个平台上`binmode`是一个“无操作”（no-op），那么这这也是一个无操作。即使没有PerlIO这一层也可用。

## :encoding(ENCODING)

这一层指定Encode模块支持的所有编码，包括直接支持和间接支持。

## :locale

这一层根据本地化环境设置来解码或编码其数据。

## :raw

这个伪层会关闭在它下面将数据解释为非二进制数据的所有层。如果在一个平台上`binmode`是一个“无操作”（no-op），那么这这也是一个无操作。即使没有PerlIO这一层也可用。

## :std

`:std`层实际上并不是一个层。导入这个层会对标准文件句柄应用指定的其他层。如果有OUT，会在标准输出和错误上设置层。对于IN，则是在标准输入上设置层。

:utf8

这个层把数据当作字符串，将其解码或编码为UTF-8。与这个层相反的是:bytes。

如果在输入流上使用内置utf8层，要做好准备处理编码错误，这一点很重要。最好的方法如下：

```
use warnings FATAL => "utf8"; # 以防出现输入编码错误
```

如果有问题，这样就可以得到一个异常。尽管可以从编码错误恢复，但是很有难度。

## ops

```
Perl-M-ops=system ... # 禁用"system"操作码
```

ops pragma禁用某些操作码，这有不可逆的全局影响。在运行Perl代码之前，Perl解释器总是把Perl源代码编译为一种内部的操作码表示。默认地，对于Perl将运行哪些操作码没有任何限制。禁用操作码只是限制Perl会编译哪些操作码。如果代码使用了一个禁用操作码，会导致一个编译错误。不过，不要以为这会提供健壮的安全性。Opcode模块提供了更多有关操作码的细节。另外参见Safe模块（第20章），这对你来说可能是一个更好的选择。

## overload

在Number模块中：

```
package Number;
use overload "+" => \&myadd,
             "-" => \&mysub,
             "*" => "multiply_by";
```

在你的程序中：

```
use Number;
$a = Number->new( 57 );
$b = $a + 5;
```

这些内置操作符可以很好地处理字符串和数字，不过应用在对象引用上时意义不大（因为，与C或C++不同，Perl不允许指针运算）。overload pragma允许重新定义将这些内置操作应用到你设计的对象时的含义。在前面的例子中，这个pragma调用会重定义Number对象上的3个操作：加法会调用Number::myadd函数，减法会调用Number::mysub函数，另外乘法赋值操作符会调用Number类（或者它的某个基类）中的multiply\_by方法。我们说这些操作符现在已经重载（overloaded），因为它们有了另外的含义（而不是因为它们有太多的含义而负荷太重，不过很有可能会这样）。



有关重载的更多内容参见第13章。

## overloading

```
no overloading;
```

这是少有的几个主要用于关闭特性而不是打开特性的pragma之一。单独使用时，它会关闭所有重载的操作，在词法作用域的其余部分中再将其恢复到它们原来的行为。

要禁用某些特定的已重载操作，可以指定overload使用的相同的键：

```
no overloading qw("");          # 没有字符串化重载
```

要重新启用重载，可以反过来使用：

```
use overloading;                # 一切恢复
use overloading @ops;           # 只重新启用其中一部分
```

## parent

```
use parent qw(Mother Father);
```

parent pragma取代了base pragma。它会加载模块，并建立继承关系而没有%FIELDS散列魔法，它还提供了一种建立继承而不加载文件的方法。

下面的例子等价于加载两个父模块，并把它们增加到@INC，但无需显式声明@INC：

```
BEGIN {
    require Mother;
    require Father;
    push @ISA, qw(Mother Father);
}
```

这里假设每个父模块有自己的文件。如果父类不在单独的文件中，可能是因为你已经在同一个文件中给出了它们的定义，或者是已经从一个文件（作为另一个类的一部分）加载了这些类，可以使用-norequire选项只建立继承关系：

```
use parent qw(-norequire Mother Father);
```

这等价于把这些类增加到@INC：

```
BEGIN {
    push @ISA, qw(Mother Father);
}
```

## re

这个pragma可以控制正则表达式的使用。它有5个可能的调用：`taint`、`eval`和`/flags`模式（词法作用域调用）；`debug`和`debugcolor`（不是词法作用域调用）。

```
use re "taint";
# 如果$dirty被污染, $match的内容也会被污染
($match) = ($dirty =~ /^(.*)$/s);

# 允许代码内插:
use re "eval";
$pat = '(?{ $var = 1 })';      # 嵌入代码执行
/alpha{$pat}omega/;           # 不会失败, 除非有-T
                                # 而且$pat被污染

use re "/a";                   # 默认地, 每个模式
                                # 都有/a标志
use re "/msx";                 # 默认地, 每个模式
                                # 都有/msx标志

use re "debug";                # 类似于"Perl-Dr"
/^(.*)$/s;                    # 输出编译时和运行时的
                                # 调试信息
use re "debugcolor";           # 等同于"debug",
                                # 不过输出有颜色

use re qw(Debug LIST);         # 调试输出的精细控制
```

如果`use re "taint"`起作用, 而且一个污染的字符串作为正则表达式的目标, 编号正则表达式变量和`m//`操作符在列表上下文中返回的值都是被污染的。如果对污染数据的正则表达式操作不是为了抽取安全的子串, 而是要做其他转换, 这就很有用。参见第20章关于污染的讨论。

`use re "eval"`起作用时, 正则表达式可以包含执行Perl代码的断言, 形如 `(?{ ... })`, 即使正则表达式包含内插的变量。出于安全原因, 通常会禁止由于正则表达式中的变量内插而导致代码段执行: 如果一个程序会从配置文件、命令行参数或CGI形式的字段读取模式, 你肯定不希望这个程序突然开始执行任意的代码 (如果它们原来没有对这种可能性做相应设计)。这个pragma只允许内插未污染的字符串, 污染的数据会产生一个异常 (如果启用了污染检查)。参见第5章和第20章。

对于这个pragma, 并不认为预编译正则表达式 (由`qr//`操作符生成) 的内插是变量内插。不过, 构建`qr//`模式时, 如果它的任何内插字符串包含代码断言, 需要有`use re "eval"`。例如:

```
$code = '(?{ $n++ })';        # 代码断言
$str = '\b\w+\b' . $code;     # 建立要内插的字符串

$line =~ /$str/;              # t这需要use re 'eval'
```

```
$pat = qr/$str/;          # 这也需要use re 'eval'
$line =~ /$pat/;          # 不过这不需要use re 'eval'
```

标志模式`use re "/flags"`会为其词法作用域中的匹配、替换和正则表达式引号操作符启用默认的模式修饰符。例如，如果希望文件中的所有模式对其字符类（`\d`，`\w`和`\s`）使用ASCII语义：

```
while (<>) {
    use re "/a";
    if (/\\d/) { # only 0 .. 9
        print "Found an ASCII digit: $_";
    }
}
```

打开某个影响经典和POSIX字符类（`/adlu`）的模式修饰符会覆盖`locale pragma`或`unicode_strings`特性的设置。

要打开多行字符串模式，使得`^`和`$`在靠近换行符的位置匹配，而不只是匹配字符串末尾（`/m`）、匹配换行符（`/s`）和扩展模式（`/x`），可以使用：

```
use re "/msx";
```

在`use re "debug"`下，Perl编译和执行正则表达式时会发出调试消息。如果运行“debugging Perl”（编译时向C编译器传入`-DDEBUGGING`），然后在Perl的`-Dr`命令行开关下执行你的Perl程序，也能得到同样的输出。取决于模式的复杂程度，得到的输出可能很庞大。调用`use re "debugcolor"`会启用更有色彩的输出，这可能很有用，假设你的终端了解这些颜色序列。可以把`PERL_RE_TC`环境变量设置为相关`termcap(5)`属性的一个列表（用逗号分隔的）来突出显示。有关的更多细节，参见第18章。

要更多地控制调试输出，可以使用`Debug`（D大写），并提供一个要调试的程序列表。`All`等价于`use re "debug"`：

```
{
    use re qw(Debug All); # 类似于"use re 'debug'"
    ...;
}
```

要想更细粒度地控制调试，还可以试试其他选项。例如，`COMPILE`选项只输出与模式编译相关的调试语句：

```
{
    use re qw(Debug COMPILE); # 类似于'use re "debug"'
    ...;
}
```

还有很多其他选项见`re`文档。



# sigtrap

```
use sigtrap;  
use sigtrap qw(stack-trace old-interface-signals); #作用相同  
  
use sigtrap qw(BUS SEGV PIPE ABRT);  
use sigtrap qw(die INT QUIT);  
use sigtrap qw(die normal-signals);  
use sigtrap qw(die untrapped normal-signals);  
use sigtrap qw(die untrapped normal-signals  
                stack-trace any error-signals);  
  
use sigtrap "handler" => \&my_handler, "normal-signals";  
use sigtrap qw(handler my_handler normal-signals stack-trace error-signals);
```

`sigtrap pragma`会代表你安装一些简单的信号处理器，使你不用再担心这些问题。如果一个未捕获的信号可能导致你的程序不正常，比如有`END{}`块、对象析构函数或者不论程序如何终止都需要运行的其他退出时处理，这些情况下，`sigtrap pragma`就很有用。

如果程序由于一个未捕获的信号而终止，程序会立即退出而不完成清理。如果捕获这样一个信号并转换为致命异常，情况就会好些：这会退出所有作用域，它们的资源会回收，并处理所有`END`块。

`sigtrap pragma`提供了两个简单的信号处理器可以使用。一个信号处理器会提供Perl栈轨迹，另一个通过`die`抛出一个异常。或者，可以提供你自己的处理器由`pragma`安装。可以指定预定义的一组要捕获的信号；还可以提供你自己的信号列表。这个`pragma`可以有选择地为那些没有处理的信号安装处理器。

传入`use sigtrap`的参数会按顺序处理。如果遇到用户提供一个信号名或者`sigtrap`预定义信号列表中的某个信号名，会立即安装一个处理器。遇到一个选项时，这只会影响处理参数列表中后来安装的那些处理器。

## 信号处理器

这些选项会影响以后安装的信号将使用哪个处理器：

### stack-trace

这个`pragma`提供的处理器会把一个Perl栈轨迹输出到`STDERR`，然后尝试转存内核。这是默认的信号处理器。

### die

这个`pragma`提供的处理器通过`Carp::croak`调用`die`，并提供一个消息指示所捕获的信号。

### handler YOURHANDLER

`YOURHANDLER`将用作为以后安装的信号的处理器。`YOURHANDLER`可以是`%SIG`的任何合

法值。要记住，很多C库调用（特别是标准I/O调用）的功能在信号处理器中都不能保证。更糟糕的是，很难猜出C库代码中的哪些部分会从Perl代码中的哪些部分调用（另一方面，`sigtrap`捕获的很多信号相当糟糕，它们只会导致程序终止，所以尝试做点什么总没有坏处，是不是？）。

## 预定义信号列表

`sigtrap pragma`有一些要捕获的内置信号列表：

### normal-signals

这些是程序正常情况下可能遇到的信号，默认地，这些信号会导致程序终止。它们是HUP，INT，PIPE和TERM信号。

### error-signals

这些信号通常反映Perl解释器或你的程序存在一个严重的问题。这些信号包括ABRT，BUS，EMT，FPE，ILL，QUIT，SEGV，SYS和TRAP信号。

### old-interface-signals

默认地这些信号会由较老版本的`sigtrap`接口捕获，包括ABRT，BUS，EMT，FPE，ILL，PIPE，QUIT，SEGV，SYS，TERM和TRAP信号。如果没有向`use sigtrap`传递信号或信号列表，就使用这个列表。

如果你的平台没有实现预定义列表中指定的一个特定信号，会悄悄地忽略该信号名（不能因为它不存在而忽略这个信号本身）。

## sigtrap的其他参数

### untrapped

这个token对随后列出的信号取消其处理器的安装（如果这些信号已经捕获或忽略）。

### any

这个token为随后列出的所有信号安装处理器。这是默认行为。

### signal

看上去像信号名的参数（也就是说与模式`/^[A-Z][A-Z0-9]*$/`匹配）会请求`sigtrap`来处理该信号。

### number

一个数值参数，要求`sigtrap pragma`的版本号至少为`number`。这就像大多数常用模块有一个`$VERSION`包变量一样：

```
% Perl-Msigtrap -le 'print $sigtrap::VERSION'
```

## sigtrap例子

为老接口信号提供一个栈轨迹：

```
use sigtrap;
```

与前面相同，不过更明确：

```
use sigtrap qw(stack-trace old-interface-signals);
```

只为4个列出的信号提供栈轨迹：

```
use sigtrap qw(BUS SEGV PIPE ABRT);
```

收到一个INT或QUIT信号时退出：

```
use sigtrap qw(die INT QUIT);
```

遇到HUP，INT，PIPE或TERM时退出：

```
use sigtrap qw(die normal-signals);
```

遇到HUP，INT，PIPE或TERM时退出，不过对于已经在程序中其他地方捕获或忽略的信号不会改变其行为：

```
use sigtrap qw(die untrapped normal-signals);
```

接收到当前未捕获的任何normal-signal时会退出。另外，接收到任何error-signal时提供一个栈轨迹：

```
use sigtrap qw(die untrapped normal-signals
               stack-trace any error-signals);
```

安装例程my\_handler作为normal-signal的处理器：

```
use sigtrap "handler" => \&my_handler, "normal-signals";
```

安装my\_handler作为normal-signals的处理器；接收到任何error-signal时提供一个Perl栈轨迹：

```
use sigtrap qw(handler my_handler normal-signals
               stack-trace error-signals);
```

## sort

v5.8之前，快速排序是Perl内置sort函数的默认算法。快速排序算法的最坏情况性能是二次的，而且不一定保证相同元素的顺序在排序前后不变（所以它是不稳定的）。Perl v5.8



将默认算法改为归并排序，它的最坏情况性能为 $O(N \log N)$ ，而且可以保证相同元素的顺序不变（所以它是稳定的）。

`sort pragma`允许你选择要使用哪个算法。如果将来哪一天默认算法不再是归并排序，你可以选择一个`stable`排序而不用选择具体的算法：

```
use sort "stable";           # 保证稳定性
use sort "_quicksort";       # 使用快速排序算法
use sort "_mergesort";       # 使用归并排序算法
use sort "defaults";         # 还原为默认行为
no sort "stable";            # 稳定性不重要

use sort "_qsort";           # 快速排序的别名

my $current;
BEGIN {
    $current = sort::current(); # 指定流行的算法
}
```

## strict

```
use strict;                  # 安装所有3种严格约束

use strict "vars";          # 变量必须预声明
use strict "refs";          # 不能使用符号引用
use strict "subs";          # 裸字符串必须加引号

use strict;                  # 安装所有限制...
no strict "vars";           # ...然后取消一个

use v5.12;                   # v5.12.0或以后版本中是默认的
```

这是一个词法作用域`pragma`，会改变Perl用来判断代码是否合法的一些基本规则。有时这些限制对于非正式的程序来说看上去过于严格，比如你可能只是想写一个5行的小过滤器程序。程序越大，要求就越严格。如果你用`use`和最小版本声明了perl的最小版本为v5.12或以后版本，就会隐含地得到这些限制。

目前，有3个方面可以加以限制：`subs`、`vars`和`refs`。如果没有提供导入列表，则假设这3种限制都存在。

### strict "refs"

如果你想解引用一个字符串而不是一个引用，不论是否有意这样做，都会生成一个运行时错误。

有关的更多内容参见第8章。

```
use strict "refs";
```

```

$ref = \$foo;      # 存储"真正的" (硬)引用
print $$ref;       # 可以正常解引用

$ref = "foo";      # 存储全局 (包) 变量名
print $$ref;       # 不正确, 在strict refs下会生成运行时错误

```

符号引用在很多方面值得怀疑。即使是原本好意的程序员也很容易不小心调用符号引用，`strict "refs"`可以防范这一点。不同于真正的引用，符号引号只能指示包变量。它们不算作引用。完成工作通常还有更好的办法：不要引用全局符号表中的一个符号，可以使用一个散列作为它自己的微型符号表。这样更高效，更可读，而且不容易出错。

不过，有些合法的操作确实需要直接访问包含变量和函数名的包全局符号表。例如，你可能想检查某个给定包的`@EXPORT`列表或`@ISA`超类，但事先不知道包名，或者你可能想安装大量函数调用，它们都是同一个闭包的别名。这些情况下最能发挥符号引用的长处，不过`use strict`起作用时才能使用符号引用，首先必须取消`refs`限制：

```

# 建立一组属性存取方法
for my $methname (qw/name rank serno/) {
    no strict "refs";
    *$methname = sub { $_[0]->{ __PACKAGE__ . $methname } };
}

```

## strict "vars"

在这个限制之下，如果想访问一个变量，但以下标准均不满足，就会触发一个编译时错误：

- 由Perl自己预定义的变量，如`@ARGV`，`%ENV`和全局标点符号变量，如`$.或$_`。
- 用`our`声明（全局变量）或用`my`或`state`声明（词法作用域变量）。
- 从另一个包导入（`vars pragma`会伪装为导入，不过应该使用`our`）。
- 使用包名和双冒号包分隔符完全限定。

`local`操作符本身并不能满足`use strict "vars"`，因为尽管“`local`”有局部的意思，但这个操作符并不会改变指定的变量的全局性。实际上，它只是在运行时执行这个块期间为这个变量（或一个数组或散列中的单个元素）给出一个新的临时值。还是需要用`our`来声明全局变量，或者使用`my`或`state`声明一个词法作用域变量。不过，可以对`our`声明局部化：

```
local our $law = "martial";
```

Perl预定义的全局变量不受这些要求的限制。这适用于整个程序范围的全局变量（包`main`中类似`@ARGV`或`$_`的变量），以及各个包的变量如`$a`和`$b`，这些通常由`sort`函数使用。类似`Exporter`等模块使用的各个包的变量仍用`our`声明：

```
our @EXPORT_OK = qw(name rank serno);
```

## strict "subs"

这个限制使Perl将所有裸字处理为语法错误。裸字（bareword）（用熊的语言来讲就是“bearword”）可以是任何裸名，或者是没有其他上下文强制解释的标识符（上下文通常由附近的关键词或token强制，或者由当前单词的预声明指定）。一直以来，裸字被解释为无引号的字符串。这个限制认为这种解释不合法。如果你想把它用作为一个字符串，就要加引号。如果想把它用作为一个函数调用，则要预声明，或者使用小括号。

作为强制上下文的一个特定情况，要记住在大括号中单独出现的裸字或者出现在=>操作符左边的裸字会认为已经加了引号，所以不受这个限制的约束。

```
use strict "subs";

$x = whatever;          # 不正确：裸字错误！
$x = whatever();        # 不过，这是可以的。

sub whatever;           # 预声明函数。
$x = whatever;          # 现在这是正确的。

# 这些用法是允许的，因为=>会加引号：
%hash = (red => 1, blue => 2, green => 3);

$rednum = $hash{red};    # 可以，这里有大括号

# 不过下面这个不行：
@coolnums = @hash{blue, green};    # 不正确：裸字错误
@coolnums = @hash{"blue", "green"}; # 可以，现在加了引号
@coolnums = @hash{qw/blue green/}; # 类似
```

## subs

```
use subs qw/winken blinken nod/;
@x = winken 3..10;
@x = nod blinken @x;
```

这个pragma将参数列表中的名字预声明为标准子例程。好处在于，现在使用这些函数可以不用像列表操作符那样加括号，就好像是你自己声明的一样。这可能没有完全声明那么有用，因为它不支持原型或属性，如：

```
sub winken(@);
sub blinken(\@) : locked;
sub nod($) : lvalue;
```

由于它基于标准导入机制，use subs pragma不是词法作用域pragma，而是包作用域pragma。也就是说，声明对它所在的整个文件都是可用的，不过仅限于当前包中。不能用no subs取消这种声明。



## 线程

Perl曾经用过几种不同的线程模型。最早有老的v5.005线程（通过Threads模块提供），不过这些已经在v5.10中去除。第二种方法在v5.8中引入，是“解释器线程”（或ithreads），这会为每个新线程提供自己的Perl解释器。如果你对其他语言的线程有所了解，对于Perl的线程，你要把你原来的知识都忘掉，因为它们只是名字相同而已，内容大不相同。

要使用threads，需要一个编译有线程支持的Perl。可以检查Perl-V的输出，在编译时选项中查找类似USE\_ITHREADS的选项。还可以检查Config模块，这个模块允许你在程序中检查编译时选项：

```
use Config;
$Config{useithreads}
or die("Recompile Perl with threads to run this program.");
```

很多随操作系统发行的Perl已经提供启用了线程的Perl，因为让所有人打开它比让所有人关闭它更容易，这让一些人有些抱怨（这说明，通过重新编译Perl，让它不提供线程支持，这样可以提高一点性能）。

下面是一个简短的小例子，它启动一些线程，分派这些线程，然后启动一个最终线程，并加入这个线程。程序不会等待分派的线程结束，不过它会等待加入的线程完成。可以用一个代码引用或子例程名创建线程，或者使用threads的async函数：

```
#!/usr/bin/perl
use v5.10;

use Config;
$Config{useithreads} || die "You need thread support to run this";

use threads;

threads->create(sub {
    my $id = threads->tid;
    foreach (0 .. 10) {
        sleep rand 5;
        say "Meow from cat $id ($_)";
    }
})->detach;

for (0 .. 4) {
    my $t = async {
        my $id = threads->tid;
        foreach (0 .. 10) {
            sleep rand 5;
            say "Bow wow from dog $id ($_)";
        }
    };
    $t->detach;
    return $t;
}
```

```
};

threads->create("bird")->join;
sub bird {
    my $id = threads->tid;
    for (0 .. 10) {
        sleep rand 5;
        say "Chirp from bird $id ($_)";
    }
}
```

可以在`perlthrtut`中读到更多有关线程的内容，这是Perl线程教程。Perl提供了一种方法可以在线程间共享变量（利用`thread::shared`），另外还有一种方法可以用`Threads::Queue`模块建立共享队列。

## utf8

```
use utf8;
```

`utf8 pragma`声明词法作用域其余部分的Perl源代码使用UTF-8编码。这允许你使用Unicode字符串直接量和标识符。

```
use utf8;
my $résumé_name = "Björk Guðmundsdóttir";
{
    no utf8;
    my $mojibake = ' '; # 可能会出错
}
```

`utf8`还提供了很多其他特性，不过由于有了`Encode`模块，它们已经被废弃。

需要说明，在v5.14中，编译器没有规范化标识符，所以无法区分构成相同字形（使用分解或未分解字符）的不同方式之间的区别。关于规范化的详细信息参见第6章。我们建议你将所有Perl标识符规范化为NFC（或NFKC），以避免两个不同变量看起来相同的问题。

## vars

```
use vars qw($frobbed @munge %seen);
```

这个`pragma`曾经用于声明一个全局变量，现在由于有了`our`修饰符，已经非正式地被废弃。前一个声明可以使用下面的代码更好地完成：

```
our($frobbed, @munge, %seen);
```

或者甚至可以使用：

```
our $frobbed = "F";
```

```
our @munge = "A" .. $frobbed;
our %seen = ();
```

不论你使用哪一种方式，要记住，它们所指的都是包全局变量，而不是文件作用域的词法作用域变量。

## version

```
use version 0.77;

my $version = version->parse($version_string);
my $qversion = qv($other_version_string);
if ($version > $qversion) {
    say "Version is greater!";
}
```

`version`模块实际上并不是一个`pragma`，不过它看起来很像一个`pragma`，因为它的名字是全小写。在v5.10之前，`VERSION`提供了一种方法可以用`qv()`对版本加引号，并比较版本号。听上去很简单，但了解到它的任务实际是多么复杂时，你可能会质疑自己当初投身编程行业是不是有些草率。例如，如何对版本1.02、1.2和v1.2.0排序？现在Perl可以在内部完成这个工作。不过，情况仍然很混乱<sup>注3</sup>。

## vmsish

```
use vmsish;           # 所有特性

use vmsish "exit";
use vmsish "hushed";
use vmsish "status";
use vmsish "time";

no vmsish "hushed";
vmsish::hushed($hush);

use vmsish;           # 所有特性
no vmsish "time";     # 不过关闭'time'
```

`vmsish pragma`控制VMS上Perl的各种特性，所以你的程序不太像一个Unix程序而更像一个VMS程序。这些特性是词法作用域特性，所以可以在需要时启用和禁用。

## exit

在`exit`下，使用`exit 1`和`exit 0`都映射到`SS$_NORMAL`，指示一个成功的退出。在Unix模拟中，`exit 1`指示一个错误。

---

注3： 你可能喜欢David Golden的“Version numbers should be boring”。



## hushed

在hushed下，从DCL运行的一个Perl程序在不成功地退出时不会向SYS\$OUTPUT或SYS\$ERROR打印消息。这不会抑制来自Perl程序本身的消息，而只影响其词法作用域中的exit和die语句，以及在这个pragma之后Perl编译的语句。

## status

在status下，system返回值以及\$?的值使用VMS退出状态而不是模拟POSIX退出状态。

## time

利用这个特性，所有时间都是相对于当地时区，而不是默认的统一时间。

## warnings

```
use warnings;          # 等同于importing "all"
no warnings;           # 等同于unimporting "all"

use warnings::register;
if (warnings::enabled()) {
    warnings::warn("some warning");
}

if (warnings::enabled("void")) {
    warnings::warn("void", "some warning");
}

warnings::warnif("Warnings are on");
warnings::warnif("number", "Something is wrong with a number");
```

这是一个词法作用域pragma，允许灵活地控制Perl的内置警告，包括编译器发出的警告，以及来自运行时系统的警告。

原先，对于Perl如何处理你的程序中的警告，只能通过-w命令行选项或\$^W变量来控制。尽管有用，但是这种控制要么全有，要么全无。-w选项甚至会启用不是你写的模块代码中的警告，有时这对你来说会有问题，另外也会让原作者很懊恼。使用\$^W禁用或启用代码块可能都不是最优的方式，因为它只适用于执行时，而在编译时不起作用<sup>注4</sup>。另一个问题是，这个程序范围的全局变量是动态作用域变量，而不是词法作用域变量。这意味着，如果你在一个块中启用这个变量，然后从那里调用其他代码，可能会再次面临一个风险，即启用这些代码中的警告（如果开发这些代码时没有考虑这些标准）。

warnings pragma会绕过这些限制，作为一个词法作用域的编译时机制，允许更细粒度地控制哪里能够或不能触发警告。警告种类的层次结构（见图29-1）已经定义，允许彼此独立

---

注4：当然，如果没有BEGIN块的情况下。

地启用或禁用不同组的警告（具体的种类划分还是试验性的，将来可能还会改变）。可以向use或no传入多个参数来组合这些警告种类：

```
use warnings qw(void redefine);
no warnings qw(io syntax untie);
```

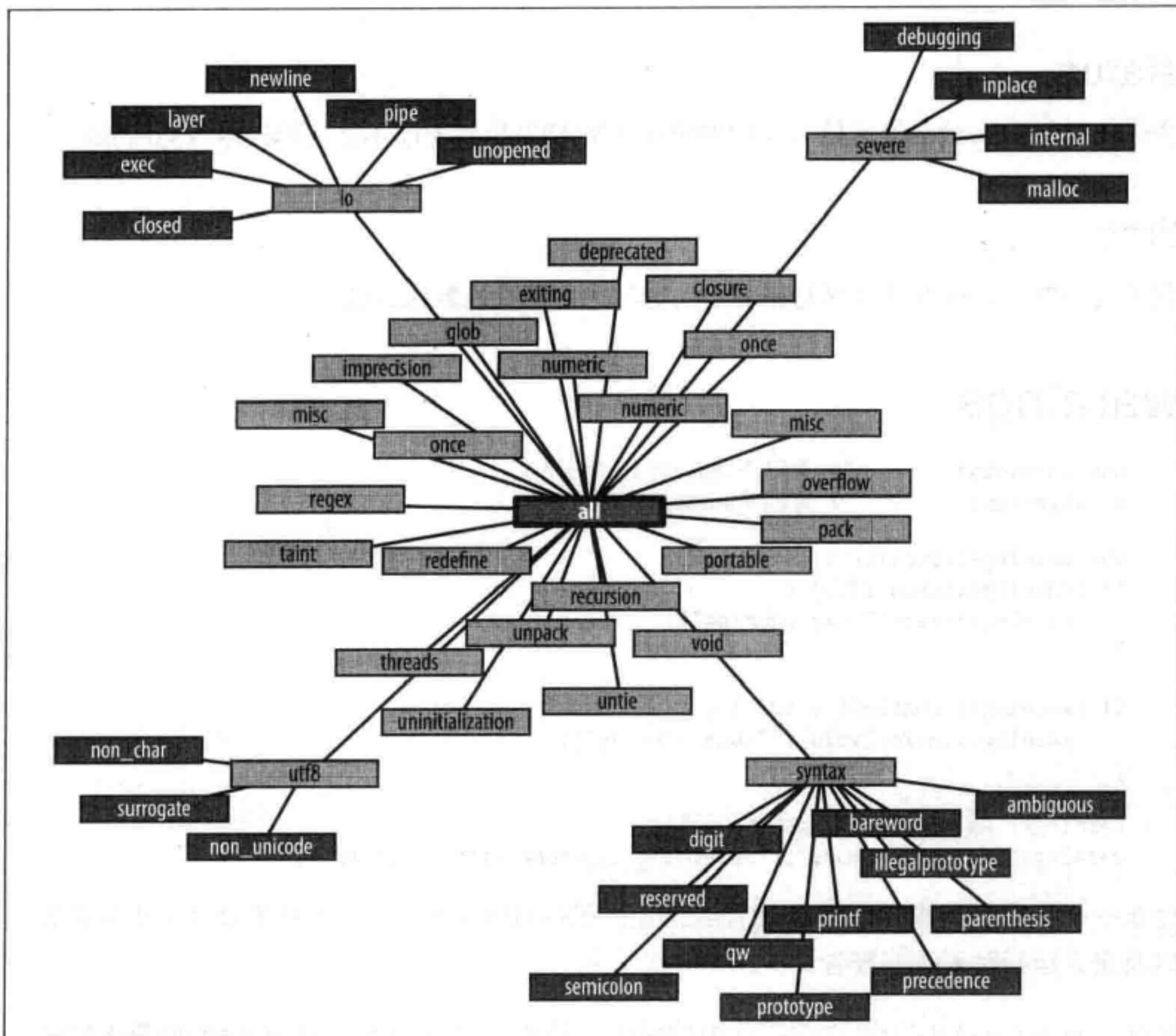


图29-1：Perl的警告类别

如果warnings pragma的多个实例在给定作用域中都生效，它们的作用会累积：

```
use warnings "void"; # 只启用"void" 警告
...
use warnings "io";   # "void" 和"io" 警告现在都启用
...
no warnings "void";  # 现在只启用"io"警告
```

要将一个特定warnings pragma启用的所有警告变成致命错误，可以在导入列表最前面使用FATAL。如果你希望一种通常只导致警告的特定情况能作为中止程序的条件，这会很有用。例如，假设你认为把一个非法的字符串用作为一个数（通常这会生成值0）不太合

适，你希望在这种情况下中止你的程序。为此，你决定，如果在需要实际字符串或数值的地方使用了未初始化的值，应该立即导致程序退出：

```
{
    use warnings FATAL => qw(numeric uninitialized);
    $x = $y + $z;
}
```

现在，如果\$y或\$z未初始化（也就是说，包含特殊标量值undef），或者如果其中某个变量包含的字符串不能清晰地转换为数值，你的程序就要异常退出。也就是说，不再是顺其自然，也不再是在启用警告时最多发出一个小小的警告做出抱怨，你的程序现在会产生一个异常（可以认为是Perl以Python模式运行）。如果不能捕获异常，这会让它成为一个致命错误。异常文本与警告消息中的文本是一样的。

如果像下面这样，在程序最上面将所有警告变成致命错误：

```
use warnings FATAL => "all";
```

这样做并不合适，因为这样无法区别编译时警告和运行时警告。编译器的第一个消息通常不需要你看到，不过如果有编译时致命警告，你可能会希望看到这个消息。更好的方法是将它们延迟到运行时再变为致命错误。

```
use Carp qw(carp croak confess cluck);
use warnings;    # 编译时警告

# 运行时，在做其他事情之前
$SIG{__WARN__} = sub { confess "FATALIZED WARNING: @_ " };
```

这种想法还有另外一个应用，就是使用cluck而不是confess。这样一来，你会得到一个栈转存，不过程序还会继续。这对于确定导致警告的代码路径很有帮助。另外参见%SIG散列中的解释来了解其他有关的例子。

warnings pragma会忽略-w命令行开关和\$^W变量值，这个pragma的设置会更优先。不过，-W命令行标志会覆盖这个pragma，启用程序中所有代码中的完全警告，甚至是用do、require或use加载的代码。换句话说，利用-W，Perl会假装你的程序中的所有块都有一个use warnings "all" pragma。可以认为它是Perl程序的lint(1)。不过参见B::Lint模块的在线文档。-X命令行标志则相反，它假装每个块都有no warnings "all"。

Perl提供了很多函数，可以帮助模块作者使他们的模块函数在表现得像内置函数一样（在调用者的词法作用域方面），也就是说，模块的用户可以在词法作用域中启用或禁用模块可能发出的警告：

**warnings::register**

注册当前模块名作为一类新的警告，使得模块的用户可以关闭其警告。



`warnings::enabled(CATEGORY)`

如果警告类别`CATEGORY`在调用模块的词法作用域中启用，则返回`true`。否则返回`false`。如果没有提供`CATEGORY`，则使用当前包名。

`warnings::warn(CATEGORY, MESSAGE)`

如果调用模块没有将`CATEGORY`设置为`FATAL`，则向`STDERR`打印`MESSAGE`。如果调用模块已经将`CATEGORY`设置为`FATAL`，则向`STDERR`打印`MESSAGE`，然后退出。如果没有提供`CATEGORY`，则使用当前包名。

`warnings::warnif(CATEGORY, MESSAGE)`

类似于`warnings::warn`，不过前提是启用了`CATEGORY`才有效。

## 用户自定义Pragma

Perl v5.10增加了一种方法，从而可以很容易地创建你自己的词法作用域`pragma`。`%H`散列包含有一些信息，其他代码可以参考这些信息来得到提示，了解你想要做什么，另外`caller`有一个引用，对于你请求的级别，这个引用会指示相应的散列版本：

```
my $hints = ( caller(1) )[10];
```

这是一个包含简单值的简单散列。不用介绍复杂的细节，只要知道这个散列会由线程共享，所以内核用一种紧凑的形式存储这个散列，排除整数、字符串和`undef`以外的所有其他值。这样很好，因为实际上只需要它指示`pragma`的特性是开还是关。这个散列也有词法作用域，所以每个词法作用域都有自己的版本。

要创建你自己的`pragma`，需要定义3个子例程`import`、`unimport`和`in_effect`。前两个会由`use`和`no`隐式调用。一般地，`use`通过调用`import`打开一个特性，而`no`通过调用`unimport`关闭该特性。除了你想要的特殊处理外，`import`和`unimport`会在`%H`中设置一个标志。在你的`pragma`之外，其他代码可以调用`in_effect`查看你的`pragma`是否启用，可以在`%H`中查找你设置的值来进行处理。

对于在`%H`中能放什么值并没有相应的规则，不过要记住，其他`pragma`也使用这个散列来完成它们自己的工作，所以要选择其他代码不太可能使用的一个键，如你的包名。

下面给出一个简单的`pragma`，它把内置`sqrt`函数替换为一个可以处理负数的函数（很原始）。`use complex`调用一个`import`方法，它在`%H`中将`complex`键设置为1，并创建一个名为`sqrt`的子例程，它使用的定义与`complex::complex_sqrt`相同。`complex_sqrt`使用`in_effect`来查看是否使用一个负数。如果是负数，则取其绝对值的平方根。如果平方小于0，则为结果追加“i”：

```
use utf8;
use v5.10;
```

```

package complex;
use strict;
use warnings;
use Carp;

sub complex_sqrt {
    my $number = shift;
    if (complex::in_effect()) {
        my $root = CORE::sqrt(abs($number));
        $root .= "i" if $number < 0;
        return $root;
    }
    else {
        croak("Can't take sqrt of $number") if $number < 0;
        CORE::sqrt($number)
    }
}

sub import {
    $^H{complex} = 1;
    my($package) = (caller(1))[0];
    no strict "refs";
    *{ "${package}::sqrt" } = \&complex::complex_sqrt;
}

sub unimport {
    $^H{complex} = 0;
}

sub in_effect {
    my $hints = (caller(1))[10];
    return $hints->{complex};
}

1;

```

现在你的程序可以创建虚数了：

```

use utf8;
use v5.10;
use complex;

say "1. √-25 is " => sqrt(-25);
say "2. √36 is " => sqrt( 36);

eval {
    no complex;

    say "3. √-25 is " => sqrt(-25);
    say "4. √36 is " => sqrt( 36);
}or say "Error: $@";

```

`no complex`取消设置`$^H{complex}`，禁止在作用域的其余部分为`complex`提供负数参数。`%^H`散列是词法作用域变量，所以会在作用域退出时恢复它原来的值。在`eval`中，`no complex`关闭了特殊处理，所以`sqrt(-25)`会产生一个错误：

```
1.  $\sqrt{-25}$  is  $5i$   
2.  $\sqrt{36}$  is 6  
Error: Can't take sqrt of -25 at sqrt.pl line 10
```

这只是一个玩具型例子，实际上，使用和返回`Math::Complex`也可以做到，甚至可以做得更好（不过那是直接使用函数，而不是把它隐藏在一个`pragma`后面）。



# 术语表

如果一个单词或短语显示为斜体，这通常表示在术语表中的另外某个位置提供了它的定义。可以把它们看作是超链接。

## accessor methods (存取方法)

用来间接查看或更新一个对象状态（其实例变量）的方法。

## actual arguments (实参)

调用函数或子例程时，为函数或子例程提供的标量值。例如，调用`power("puff")`时，字符串`"puff"`就是实参。另外参见参数（argument）和形参（formal arguments）。

## address operator (地址操作符)

有些语言直接处理值的内存地址，不过这就像玩火，很危险。Perl提供了一组石棉手套来处理所有内存管理。Perl中与地址操作符最接近的是反斜线操作符，不过这会提供一个硬引用，这比内存地址要安全得多。

## algorithm (算法)

一个明确的步骤序列，有清晰的解释，以便计算机完成任务。

## alias (别名)

某个东西的绰号，在各个方面都表现得好像你使用的是原来的名字而不是这个绰号。foreach循环的循环变量中、map或grep操作符的\$\_变量中、sort比较函数的\$a和\$b中，以及子例程调用实参的@\_的各

个元素中都可能会隐式创建临时别名。可以通过导入符号或者通过对类型图赋值在包中显式地创建永久别名。包变量的词法作用域别名由our声明显式创建。

## alphabetic (字母)

单词中放入的字符。在Unicode中，这是包括所有字形和某些符号、字母数字（如罗马数字）和各种组合标记的所有字母。

## alternatives (候选项)

一个可选项的列表，只能从中选择一项，类似于“你喜欢A门、B门还是C门？”正则表达式中的候选项用一个单竖线（|）分隔。正常的Perl表达式中的候选项用一个双竖线（||）分隔。布尔表达式中的逻辑候选项用||或or分隔。

## anonymous (匿名)

用于描述一个指示对象（referent）无法通过一个命名变量直接访问。这种指示对象必须通过至少一个硬引用（hard reference）间接访问。最后一个硬引用消失后，就会立即撤销这个匿名指示对象。

## application (应用)

一个更大更复杂的程序（program），往往有一个更特别的名称，使人们不会意识到他们使用的是一个程序。

## architecture (体系结构)

你使用的计算机的类型，这里的计算机“类型”表示所有这些计算机都共享一个兼容的机器语言。由于Perl程序（通常是简单的文本文件，而不是可执行的映射，与其他语言（如C）编写的程序（这些程序会编译为机器码）相比，Perl程序不太受其所在体系结构的影响。另外参见平台（platform）和操作系统（operating system）。

## argument (参数)

提供给程序、子例程、函数或方法的一个数据，告诉它要做什么。也称为“parameter”。

## ARGV

包含命令行参数向量的数组名。如果使用空`<>`操作符，ARGV既是用于遍历参数的文件句柄，又是包含当前输入文件名的标量。

## arithmetical operator (算术操作符)

这是类似+或/的符号，告诉Perl完成你应该在小学就学过的算术运算。

## array (数组)

一个有序的值序列，采用一种便于访问的方式存储，可以使用一个整数下标来访问各个值（下标指定了值在这个序列中的偏移量）。

## array context (数组上下文)

这是原先的一种说法，更准确的说法是列表上下文（list context）。

## Artistic License (Artistic权限)

这是Larry Wall为Perl创建的一个开源权限，从而最大限度地提高Perl的有用性、可用性和可修改性。当前的版本是2.0 (<http://www.opensource.org/licenses/artistic-license.php>)。

## ASCII

美国标准信息交换码（The American Standard Code for Information Interchange），这是一个7位字符集，只能用来表示英语文本。通常不太严格地用来描述各种ISO-8859-X字符集的低128个值，这是一组互相不兼容的8位码，最好描述为半ASCII。另外参见Unicode。

## assertion (断言)

正则表达式的一个组成部分，对于要匹配

的模式必须为true，但本身不一定匹配任何字符。通常特别用来表示一个0宽度的断言。

## assignment (赋值)

这个操作符的任务就是改变一个变量的值。

## assignment operator (赋值操作符)

可以是一个常规赋值或一个组合操作符（由一个普通赋值和另外某个操作符组成），可以原地改变一个变量的值，也就是说，相对于它原来的值进行调整。例如，`$a += 2`会把2增加到\$a。

## associative array (关联数组)

请参见散列。关联数组这个词是Perl4原先用来表示散列的说法。一些语言把它叫做字典。

## associativity (结合性)

如果有“A 左操作符 B 右操作符 C”而且两个操作符有相同的优先级时，结合性会确定先完成左操作符还是右操作符。类似+的操作符是左结合的，而像\*\*等操作符是右结合的。参见第3章，其中给出了操作符及其结合性的一个列表。

## asynchronous (异步)

这是指相对时间顺序不确定的事件或活动，因为同时会发生太多事情。因此，异步事件就是你不知道何时发生的事件。

## atom (原子)

这是正则表达式的一个组成部分，可能与包含一个或多个字符的子串匹配，由后面的量词（quantifier）看作是一个不可分的语法单位（与断言相反，断言匹配零宽度的串，不能加量词）。

## atomic operation (原子操作)

德谟克利特把这种不可分的物质起名为“原子”时，他的意思是这种东西不能切割：ἀ-（不）+τομος（可切割）。原子操作是指不可中断的动作（不过无核区不会禁止这么做）。

## attribute (属性)

这是一个新特性，允许变量和子例程的声明带修饰符，如`sub foo : locked method`。这也是对象的实例变量的另一个名字。

## autogeneration (自动生成)

这是对象操作符重载的一个特性，利用自



动生成，某些操作符的行为可以使用更基本的操作符合理地推导得出。这里假设重载的操作符通常与常规操作符有相同的关系。参见第13章。

#### autoincrement (自增)

将一个变量自动加1，所以得名++操作符。要自动地将一个变量减1，则称为“自减”。

#### autoload (自动加载)

按需加载（也称为“懒”加载）。具体地，是指代表一个未定义的子例程调用一个AUTOLOAD子例程。

#### autosplit (自动分解)

自动地分解一个字符串，类似于-p或-n下模拟awk的-a开关的工作（参见AutoSplit模块，它与-a开关没有任何关系，不过与自动加载有很大关系）。

#### autovivification (自动生成)

这是一个希腊罗马词汇，表示“重生”。在Perl中，存储位置（左值）根据需要自发地自行生成，包括创建硬引用值指向下一层存储位置。赋值\$a[5][5][5][5][5]="quintet"可能会创建5个标量存储位置，另外4个引用（在前4个标量位置中）分别指向4个新的匿名数组（包含后4个标量位置）。不过自动生成的关键其实是无需你操心这一点。

#### AV

“数组值”（array value）的缩写，表示Perl的一个内部数据类型，用来保存一个数组。AV类型是SV的一个子类。

#### awk

顾名思义，这个术语含义很清楚，它就是“awkward”的简写。恰好也表示一个非凡的文本处理语言，Perl从中汲取了一些先进的思想。

#### backreference (反向引用)

正则表达式中无修饰小括号中一个子模式捕获的子串。同一个模式中后面加反斜线的十进制数（\1、\2等）可以反向引用到当前匹配中相应的子模式。在这个模式之外，编号变量（\$1、\$2等）可以继续指示这些值（只要这个模式是当前动态作用域的最后一个成功的匹配）。

#### backtracking (回溯)

这是指“如果必须反复地做一件事，我会

用不同的方式去做”，实际上就是返回去，采用不同的方法重做一次。从数学角度来讲，这是指在一个可能性树中从一次不成功的递归返回。Perl尝试匹配一个有正则表达式的模式时，如果之前的所有尝试都不成功，就会回溯。另外参见第5章“小引擎/能（与不能）/”一节。

#### backward compatibility (向后兼容性)

表示你仍然可以运行你的老程序，因为我们没有破坏它依赖的特性或bug。

#### bareword (裸字)

这是一个不明确的词，所以在use strict 'subs'下会认为是非法的。如果没有这个限制，会认为已经为裸字加了引号。

#### base class (基类)

一个通用的对象类型；也就是说，这是一个类，其他更特定的类会通过继承派生这个类。也被一些尊敬祖先的人称为“超类”。

#### big-endian (大端)

源自斯威夫特的《格列佛游记》：有人吃鸡蛋时先吃大头一端。也指一些计算机把一个字的高位字节存储在低字节地址而不是低位字节。通常认为大端机器优于小端。另外参见小端。

#### binary (二进制)

必须以二进制处理数字。这说明实际上只有两个数：0和1。也用于描述一个“非文本”的文件，可能因为这种文件在其字节中使用了所有二进制位。随着Unicode的出现，这个区别已经值得怀疑，越来越失去意义。

#### binary operator (二元操作符)

有两个操作数的操作符。

#### bind (绑定)

为一个套接字指定一个特定的网络地址。

#### bit (二进制位)

0到1范围内的一个整数（包括0和1）。这是最小的信息存储单位。这是字节的八分之一或“一块钱”（“八块钱”（Pieces of Eight）一词的出处是，西班牙银圆可以分为8份，每一份仍然可以当钱用。正是这个原因，现在有时也称25分钱为“两位”）。

#### bit shift (移位)

在一个计算机字中将位左移或右移，其效



果为按相应位数乘以或除以2的一个幂。

### bit string (位串)

一个位序列，实际上曾经就被认为是位序列。

### bless (祝福)

在公司里，正式批准一件事时，比如“工程部副总裁祝福我们的WebCruncher项目取得成功”。类似地，在Perl中，也要正式批准一个指示对象，使它能够作为一个对象，如WebCruncher对象。参见第27章中的bless函数。

### block (阻塞或块)

一个进程必须等待时所做的事情：“我的进程被阻塞，在等待磁盘操作完成。”作为一个与此毫不相关的名词，它还指示一大块数据，其大小为操作系统可以处理的大小（通常是2的幂，如512或8192）。一般是指来自或放在磁盘文件上的一个数据块。

### BLOCK (代码块)

这是一个语法构造，由一系列Perl语句组成，用大括号定界。例如，if和while语句就定义为代码块。有时，我们也称“代码块”表示一个词法作用域。也就是说，相当于一个代码块的一个语句序列，如在eval或文件中，尽管这些语句没有用大括号定界。

### block buffering (块缓冲)

这是一种提高输入输出效率的方法，一次传递一个块。默认地，Perl会对磁盘文件完成块缓冲。另外参见缓冲区和命令缓冲。

### Boolean (布尔)

只能是true或false的一个值。

### Boolean context (布尔上下文)

一种特殊的标量上下文，在条件中使用来确定一个表达式返回的标量值是true还是false。不要计算为字符串或数字。另外参见上下文。

### breakpoint (断点)

程序中的一个位置，你告诉调试器要在这里停止执行，以便你查看是否出现问题。

### broadcast (广播)

同时向多个目标发送一个数据报。

### BSD

一种精神鸦片，在80年代非常流行，可

能是在加州大学伯克利分校或者附近地区开发。在很多方面与名为“System V”的处方药很类似，不过更有用（或者，至少更有趣）。全名是“Berkeley Standard Distribution”（伯克利标准发行版）。

### bucket (桶)

一个包含（可能）多个记录的散列表中的一个位置，在这个散列中，根据其散列函数，键“杂乱地”地与相同的散列值对应（作为内部策略，不用担心这一点，除非你想深入内部）。

### buffer (缓冲区)

保存数据的临时位置。块缓冲的数据表示只要缓冲区满，数据就会传递到它的目标位置。行缓冲表示只要接收到一个完整的行就会传递数据。命令缓冲是指每次完成一个print命令（或等价命令）时都会传递数据。如果你的输出未缓冲，系统会一次一个字节地进行处理，而不使用一个存储区。这样效率可能相当低。

### built-in (内置)

Perl语言中预定义的函数。即使由于重载而被隐藏，也可以利用CORE::伪包通过限定其名字来访问内置函数。

### bundle (包)

CPAN上一组相关的模块（有时也指分组到一个开关簇的一组命令行开关）。

### byte (字节)

一段数据，大多为8位。

### bytecode (字节码)

一种洋泾滨语，人们不想暴露他们的方位时就会用这个词（参见大端）。这个名字来自于20世纪末编译器和解释器所讲的一些类似的语言（出于类似的原因）。这些语言的特点就是把所有一切都表示为一个不依赖于体系结构的字节序列。

### C

一种因其由内而外的类型定义、神秘莫测的优先级规则以及函数调用机制的强大能力而为人们所喜爱的语言（嗯，实际上，人们最初转向C只是因为他们发现小写的标识符比大写标识符更好读）。Perl是用C写的，所以毫不奇怪Perl从C借用了很多想法。

### cache (缓存)

数据存储库。不是多次计算开销很大的答

案，而是只计算一次，把结果保存起来。

#### callback (回调)

这是你为程序中另外某个部分注册的一个处理器，当你感兴趣的事件发生时，希望程序的那个部分会触发你的处理器。

#### call by reference (按引用传递)

这是一种参数传递机制，形参直接指示实参，子例程可以通过改变形参来改变实参。也就是说，形参是实参的一个别名。另外参见按值传递。

#### call by value (按值传递)

这是一种参数传递机制，形参指示实参的一个副本，子例程不能通过改变形参来改变实参。另外参见按引用传递。

#### canonical (规范)

缩减为一种标准形式以利于比较。

#### capture variables (捕获变量)

这些变量包含模式匹配中所记住的文本（如\$1和\$2，%+和%-）。另外参见第5章。

#### capturing (捕获)

在正则表达式中对一个子模式使用小括号，将匹配的子串存储为一个反向引用（捕获字符串在列表上下文中也作为列表返回）。另外参见第5章。

#### cargo cult (船货崇拜)

只是复制和粘贴代码而不知其所以然，只是迷信地相信它的价值。这个词源于土著文化对探险家和殖民者先进技术文化的盲目崇拜。另外参见*Gods Must Be Crazy*。

#### case (大小写)

某些字符的一种属性。原先排版机将大写字母存储为大小写中较大的形式，而小写字母为较小的形式。Unicode则识别3种大小写：小写（字符属性\p{lower}）、标题形式（\p{title}）和大写（\p{upper}）。第4种大小写映射称为*foldcase*，它本身并不是一种单独的大小写，只是在内部用来实现大小写转换。并不是所有字母都有大小写，另外一些非字母也有大小写。

#### casefolding (大小写转换)

不区分大小写完成字符串比较和匹配。在Perl中，这是用/i模式修饰符、fc函数和\F双引号转换转义实现的。

#### casemapping (大小写映射)

将一个字符串转换为4种Unicode大小写形式之一的过程。在Perl中，这是使用fc、lc、ucfirst和uc函数实现的。

#### character (字符)

字符串的最小单个元素。计算机将字符存储为整数，不过Perl允许将字符作为文本来处理。用来表示一个特定字符的整数称为该字符的码点。

#### character class (字符类)

用中括号括起的字符列表，在正则表达式中用于指示这个集合中的某个字符可能出现在给定位置。不严格地讲，任何预定义的字符集都可以用作字符类。

#### character property (字符属性)

可以与\p或\P元符号匹配的预定义字符类。Unicode为每一个可能的码点定义了数百个标准属性，Perl也定义了自己的一些字符属性。

#### circumfix operator (环缀操作符)

环绕其操作数的操作符，如尖括号操作符或小括号，或者是个拥抱。

#### class (类)

用户自定义的类型，Perl中通过包来实现，包（直接或通过继承）提供方法（也就是子例程）来处理类的实例（其对象）。另外参见继承。

#### class method (类方法)

这种方法的调用者是一个包名，而不是一个对象引用。与类关联的方法。另外参见实例方法。

#### client (客户端)

在网络编程中，客户端进程向一个服务器进程建立联系，来交换数据或者可能还会接收某个服务。

#### closure (闭包)

这是一个匿名子例程，运行时生成它的一个引用时，会跟踪外部可见的词法作用域变量的身份，即使在这些词法作用域变量超出作用域之后仍会跟踪。之所以称之为“闭包”就是因为这种行为让数学家有一种闭包的感觉。

#### cluster (簇)

加小括号的子模式，用来把正则表达式的某些部分分组到一个原子中。



## CODE

将ref函数应用到一个子例程的引用时，会返回这个词，参见CV。

## code generator (代码生成器)

用一种底层语言写代码的系统，如生成编译器后端的代码。另外参见程序生成器。

## codepoint (码点)

计算机用来表示一个给定字符的整数。ASCII码点在0到127范围内；Unicode码点在0到0x1F\_FFFF范围内；Perl码点在0到 $2^{32}-1$ 或0到 $2^{64}-1$ ，这取决于你的本地整数大小。在Perl文化中，码点有时称为序数。

## code subpattern (代码子模式)

这是一个正则表达式子模式，其真正用途是执行一些Perl代码，例如`(?{...})`和`(??{...})`子模式。

## collating sequence (比较序列)

字符排序的顺序。由字符串比较例程用来确定字符的先后顺序，比如这个“比较序列”应该放在这个术语表的哪个位置。

## co-maintainer (合作维护者)

有权限在PAUSE中对命名空间建索引的人。任何人都可以上传任何命名空间，不过只有主要维护者和合作维护者可以建索引。

## combining character (组合字符)

任何通用类别为组合标记(`\p{GC=M}`)的字符，可能是空格或非空格。有些甚至是不可见的。一个组合字符序列再加一个字形基字符可以共同构成一个用户可见的字符，称为字形。大多数(但不是全部)变音符号都是组合字符，反之亦然。

## command (命令)

在shell编程中，这是程序名及其参数的语法组合。更宽松地讲，就是你在shell(一个命令解释器)中输入用来完成某个任务的语句。甚至还可以进一步放宽，这是一个Perl语句，以一个标签开头，通常有一个分号结尾。

## command buffering (命令缓冲)

Perl中的一种机制，允许你存储各个Perl命令的输出，然后请求操作系统刷新输出。这是通过将`$|($AUTOFLUSH)`变量设置为一个true值来启用的。如果你不希望数据保留在原地(这是有可能的，因为文件或管道上会默认使用块缓冲)，而希望它们前往

本该去的地方，就可以使用命令缓冲。

## command-line arguments (命令行参数)

告诉shell执行一个命令时随程序名提供的值。这些值会通过`@ARGV`传递到Perl程序。

## command name (命令名)

在命令行中键入的当前执行的程序的名字。在C中，命令名会作为第一个命令行参数传入程序。在Perl中，它作为`$0`单独传递。

## comment (注释)

不影响程序含义的标注。在Perl中，注释由一个`#`字符引入，一直持续到行尾。

## compilation unit (编译单元)

当前编译的文件(或字符串，如果在eval中)。

## compile (编译)

将源代码转换为一种机器可用形式的过程，参见编译阶段。

## compile phase (编译阶段)

Perl开始运行主程序的阶段。另外参见运行阶段。编译阶段主要用于编译时，不过计算BEGIN块、use声明或常量子表达式时也用于运行时。use声明的启动和导入代码也会在编译阶段运行。

## compiler (编译器)

严格地讲，这个程序吞进一个程序，再吐出另一个文件，它包含这个程序的一种“更可执行”的形式，通常由原生机器指令组成。根据定义来讲，Perl程序并不是一个编译器，不过它确实包含一个编译器，可以在perl进程本身将一个程序转换为更可执行的形式(语法树)，解释器再解释这个语法树。不过，还有一些扩展模块可以让Perl表现得更像一个“真正的”编译器。参见第16章。

## compile time (编译时)

在编译时，Perl要为代码赋予含义，这与运行时不同，在运行时，Perl认为它知道你的代码表示什么，只是去做它认为你的代码要做的事情。

## composer (生成器)

如果指示对象并不是一个对象(object)，如一个匿名数组或散列(或者一个奏鸣曲)，生成器就是这样一个指示对象的“构造函数”。例如，一对大括号相对于



一个散列的生成器，一对中括号相当于一个数组的生成器。参见8章“创建引用”一节。

#### concatenation (联接)

把一只猫鼻子与另一只猫尾巴连在一起的过程。对两个字符串也有类似的操作。

#### conditional (条件)

“如果……”另外参见布尔上下文。

#### connection (连接)

在电话领域，这表示打电话的人和接电话的人的电话之间的一个临时电路。在网络中，客户端和服务端之间也有类似的临时通路。

#### construct (构造)

作为名词，这是指一段语法，由更小的语法单位组成。作为一个及物动词，是指用一个构造函数创建一个对象。

#### constructor (构造函数)

生成、初始化、祝福和返回一个对象的任何类方法、实例方法或子例程。有时我们不严格地使用这个词表示一个生成器。

#### context (上下文)

外围环境。外围代码给定的上下文确定了一个特定表达式要返回何种类型的数据。有3种主要的上下文，分别是列表上下文、标量上下文和void上下文。标量上下文有时进一步划分为布尔上下文、数值上下文、字符串上下文和void上下文。还有一种“不关心”上下文（如果你感兴趣，参见第2章）。

#### continuation (连续)

把多个物理行处理为一个逻辑行。对于Makefile行，在换行符前面放一个反斜线就可以将它们续行。邮件首部（由RFC 822定义）通过在换行符后面加一个空格或tab来续行。一般地，Perl中的行不需要任何形式的连续标志，因为空白符（包括换行符）会直接忽略。一般是这样。

#### core dump (核心转存)

这是进程的残骸，以文件形式放在进程的工作目录中，通常是某种致命错误导致的结果。

#### CPAN

Perl综合典藏网（The Comprehensive Perl Archive Network）。详细信息参见前言和

第19章。

#### C preprocessor (C预处理器)

典型的C处理器的第一趟处理，会处理以#开头的行作为条件编译和宏定义，另外根据当前定义对程序文本完成各种处理。又名为cpp(1)。

#### cracker (黑客)

破坏计算机系统安全的人。黑客可能是一个真正的黑客，也可能只是一个刚开始写脚本的小孩子。

#### currently selected output channel (当前选择的输出通道)

用select(FILEHANDLE)指定的最后一个文件句柄（filehandle）。如果没有选择文件句柄，则为STDOUT。

#### current package (当前包)

编译当前语句所在的包。在程序文本中，通过当前词法作用域或任何外围词法作用域向上扫描直到找到一个包声明。这就是你的当前包的名字。

#### current working directory (当前工作目录)

参见工作目录。

#### CV

在学术界，这是指一种简历。在Perl中，这是包含一个子例程的内部“代码值”类型定义。CV类型是SV的一个子类。

#### dangling statement (悬空语句)

一个裸语句，没有任何大括号，直接作为一个if或while条件。C允许有这种语句，但Perl不允许。

#### datagram (数据报)

一个数据包，如UDP消息，（从程序的角度看）可以在网络上独立地发送（实际上，所有数据报都会在IP层独立地发送，不过类似TCP的流协议会对程序隐藏这一点）。

#### data structure (数据结构)

不同数据彼此之间如何关联，以及将它们放在一起时是什么形状，如一个矩形表或三角形树。

#### data type (数据类型)

一组可能的值，以及知道如何处理这些值的所有操作。例如，一个数值数据类型会有一组可以处理的数字，以及在这些数字上完成的各种数学操作，不过这些操作对

于“Kilroy”之类的字符串没有太大意义。字符串有其自己的操作，如联接。组合类型由多个更小的部分组成，通常有组合和分解这些部分的操作，可能还有重排操作。对真实世界事物建模的对象通常会有与真实活动对应的操作。例如，如果你对一个电梯建模，你的电梯对象可能有一个open\_door（开门）方法。

## DBM

表示“数据管理”例程，这是使用磁盘文件模拟一个关联数组的一组例程。这些例程使用一个动态散列机制，只需要两次磁盘存取就可以定位任何记录。DBM文件允许Perl程序在多次调用之间保持一个持久的散列。可以将你的散列变量绑定到不同的DBM实现。

## declaration（声明）

这是一个断言（assertion），指出某个东西存在，可能会描述它的样子，但不规定如何使用或者在哪里使用。声明就像菜谱里的一部分，“两杯面粉，一个大鸡蛋，4到5勺……”与它相对的是语句。需要说明，有些声明也相当于语句。如果子例程声明提供了体，也可以作为定义。

## declarator（声明符）

告诉你的程序你想要哪种类型的变量。Perl不要求你声明变量，但是可以使用my、our或state来指示不同于默认设置的变量。

## decrement（减1）

从一个变量减去一个值，如“decrement \$x”（表示从其值减去1）或“decrement \$x by 3”。

## default（默认值）

如果没有提供你自己的值，这是为你选择的默认值。

## defined（已定义）

有某种含义。Perl认为人们想做的某些事情是没有意义的。具体地，使用根本不会给定一个值的变量，以及对不存在的数据完成某些操作，这些就没有任何意义。例如，如果你想读取超过文件末尾的数据，Perl会发回一个未定义的值。另外参见false和第27章中的defined。

## Delimiter（定界符）

这是一个字符或字符串，可以为一个任意大小的文本对象设置界限，不要与分隔符

或终止符混淆。“定界”实际上表示“包围”或“闭合”（比如小括号的作用）。

## dereference（解引用）

这是一个有趣的计算机学术语，表示“沿着一个引用到达它指向的对象”。“解”（de）的含义是指要经过一个间接层。

## derived class（派生类）

对于一个更通用的类（称为基类），这个类定义了它自己的一些方法。需要说明，类并不严格地分为基类或派生类：一个类可以同时作为派生类和基类，这很普遍。

## descriptor（描述符）

参见文件描述符。

## destroy（撤销）

释放一个指示对象的内存（首先触发其DESTROY方法，如果有的话）。

## destructor（析构函数）

这是一个特殊的方法，对象撤销自己时就会调用这个方法。Perl程序的DESTROY方法并不完成具体的撤销工作。Perl只是触发这个方法，因为类有可能想要完成一些相关的清理工作。

## device（设备）

连接到计算机的一个硬件设备（如磁盘、磁带驱动器、调制解调器、游戏杆或鼠标等），操作系统努力使它看上去像一个文件（或一组文件）。在Unix下，这些假文件通常在/dev目录下。

## directive（指令）

pod指令。另外参见第23章。

## directory（目录）

一个特殊的文件，其中包含其他的文件。一些操作系统把它们称为“文件夹”、“抽屉”、“编目”或“目次”等。

## directory handle（目录句柄）

表示打开一个目录用于读取的特定实例的名字，直到你将其关闭。另外参见opendir函数。

## discipline（原则）

一些人需要这些，一些人想避开这些。这是原来对I/O层的一种说法。

## dispatch（分派）

将数据发送到正确的目标。一般采用比喻



手法来指示将程序控制传递到一个用算法选择的目标，通常是在一个函数引用表中查找声明符来选择，或者如果是对象方法，则是遍历继承树来查找更特定的方法定义。

#### distribution (发行版本)

软件系统的一个标准发行版本。默认情况下都隐含包括源代码。如果不是这样，可以称为一个“仅二进制”发行版本。

#### dual-lived (双重性)

一些模块既在标准库中又在CPAN上。人们修改模块版本时，这些模块会在两条线上分头开发。目前的趋势逐步清理这种情况。

#### dweomer (幻像)

一种假象或幻觉。当Perl的魔法*dwimmer*效果不如预期，看上去像是神秘幻影的产物“来自古英语”。

#### dwimmer

DWIM是“做我要做的事”(Do What I Mean)的缩写，其原则是应当做你要求它做的事情，没有任何含糊。可以做到“dwimming”的一段代码就是一个“dwimmer”。Dwimming可能需要大量幕后魔法，这(如果它没有正常在留在幕后)称为dweomer。

#### dynamic scoping (动态作用域)

动态作用域的工作在动态作用域发生，使变量在首次使用它的块的其余部分都可见，另外在块其余部调用的所有子例程中也可见。利用一个local操作符，动态作用域变量可以临时改变它们的值(以后还可以隐式地恢复)。与词法作用域对照。如果不那么严格，可以用来表示在中间调用另一个子例程的子例程如何在运行时“包含”该子例程。

#### eclectic (电气)

有很多说法。有些人可能认为说法太多了。

#### element (元素)

一个基本的构建块。谈到数组时，元素就是构成数组的一项。

#### embedding (嵌入)

一个东西包含在另外一个东西中就称为嵌入，比如，“我在我的编辑器里嵌入了完整的Perl解释器”(这可能有些奇怪)。

#### empty subclass test (空子类测试)

这个概念是指空派生类应该与其基类有相同的表现。

#### encapsulation (封装)

这个抽象的面纱将接口与实现分开(不论是否强制)，这要求对对象状态的访问都要通过方法完成。

#### endian (字节顺序)

参见小端和大端。

#### en passant (顺道)

复制一个值时改变这个值“源自法语‘顺路’，类似于国际象棋中的吃过路兵”。

#### environment (环境)

进程从其父进程继承的环境变量集合。要通过%ENV访问。

#### environment variable (环境变量)

利用这种机制，一些高层代理(如用户)可以将其首选项向下传递到其将来的后代(子进程、孙进程、曾孙进程等)。每个环境变量是一个键/值对，如散列中的一个记录。

#### EOF

文件末尾(End of File)。有时用作为here文档的终止字符串。

#### errno (错误号)

系统调用失败时返回的错误号。Perl按名\$!(或者如果使用English模块则为\$OS\_ERROR)来引用这个错误号。

#### error (错误)

参见异常或致命错误。

#### escape sequence (转义序列)

参见元符号。

#### exception (异常)

错误的一种有趣的叫法。另外参见致命错误。

#### exception handling (异常处理)

程序对错误的响应方式。Perl中的异常处理机制是eval操作符。

#### exec

抛开当前进程的程序，把它替换为另一个程序，但不退出这个进程或回收占用的任何资源(除了原来的内存映像)。



### executable file (可执行文件)

这个文件有特别的标志,告诉操作系统可以把这个文件当作一个程序来运行。通常简写为“可执行文件”(executable)。

### execute (执行)

运行一个程序或子例程(与内置的kill没有关系,除非你要运行一个信号处理器)。

### execute bit (执行位)

这个特殊标志指出可以运行这个程序。实际上Unix下有3个执行位,使用哪个执行位取决于你单独拥有文件,还是合作拥有文件,或者根本不是文件的所有者。

### exit status (退出状态)

参见状态。

### exploit (漏洞)

在这里作名词使用,表示一种已知的方法来破坏程序,让它做程序作者原本不算做的事情。你的任务是编写无漏洞的程序。

### export (导出)

使一个模块的符号可以由其他模块导入。

### expression (表达式)

需要值的地方都可以合法地放置一个表达式。通常由直接量、变量、操作符、函数和子例程调用组成,但不严格按照这个顺序。

### extension (扩展)

一个引入已编译C或C++代码的Perl模块。更一般地,可以是能够编译到Perl中的任何试验性的选项,如多线程。

### false

在Perl中,如果在一个字符串上下文中计算,任何看上去像""或"0"的值都是false。由于未定义的值计算为"",所以所有未定义的值都是false,但是并不是所有false值都是未定义的。

### FAQ

常问的问题(Frequently Asked Question),不过不一定是最常回答的问题,特别是如果答案已经出现在随Perl发布的Perl FAQ中,通常不会再重复回答。

### fatal error (致命错误)

未捕获的异常,这会导致在标准错误流打印一个消息后终止进程。eval中发生的错

误不是致命的。相反,eval先将异常消息放在\$@(\$EVAL\_ERROR)变量中,之后再终止。你可以尝试用die操作符调用一个致命错误(称为抛出或产生一个异常),不过这可能会被一个动态的外围eval捕获。如果未能捕获,这个die就变成一个致命错误。

### feeping creaturism (创作蔓延)

也称为“特性蔓延”,表示一种生理冲动想要再为程序增加一个特性。

### field (字段)

一个数值或字符串数据,这是一个更大字符串、记录或行的一部分。宽度可变的字段通常按分隔符分解(所以使用split来抽取字段),而定宽字段通常都在固定的位置(所以使用unpack)。实例变量也称为“字段”。

### FIFO

先进先出(First In, First Out)。另外参见LIFO。这也是命名管道的另外一个名字。

### file (文件)

一个命名的数据集合,通常存储在磁盘上文件系统的某个目录中。如果按办公室来考虑,这有些像文档。现代文件系统中,实际上可以为一个文件指定多个名字。一些文件有特殊的属性,如目录和设备。

### file descriptor (文件描述符)

这是一个小编号,操作系统用它来跟踪你所指的打开的文件。Perl将文件描述符隐藏在一个标准I/O流中,然后将这个流与一个文件句柄关联。

### fileglob (文件团)

对文件名的一个“通配”匹配。参见glob函数。

### filehandle (文件句柄)

表示打开一个文件的特定实例的标识符(不一定与实际的文件名相关),直到将其关闭。如果要连续打开和关闭多个不同文件,最好用相同的文件句柄分别打开各个文件,这样就不用担心必须编写单独的代码来处理各个文件。

### filename (文件名)

文件的一个名字。这个名字列在一个目录中。可以在一个open中用文件名告诉操作系统你想打开哪个文件,并把这个文件与一个文件句柄关联,这个文件句柄将在你

的程序中作为这个文件，直到你将其关闭。

### filesystem (文件系统)

位于磁盘上某个分区中的一组目录和文件。有时称为一个“分区”。可以改变一个文件的名称，或者甚至可以将文件从一个目录移到文件系统中的一个目录中，而不用真正移动文件本身，至少在Unix下可以如此。

### file test operator (文件测试操作符)

一个内置的一元操作符，用来确定关于一个文件的某个特性是否为`true`，如`-o $filename`可以测试你是否是这个文件的所有者。

### filter (过滤器)

这个程序设计为取一个输入流，将它转换为一个输出流。

### first-come

第一个上传某个命名空间的PAUSE作者可以自动成为这个命名空间的主要维护者。“first-come”权限可以区别得到此角色的主要维护者和自动接收这个权限的人。

### flag (标志)

我们尽量避免这个说法，因为这表示很多意思。它可能表示一个命令行开关，本身没有参数（如Perl的`-n`和`-p`标志），或者不太常用的，也表示一个一位指示符（如`sysopen`中使用的`O_CREAT`和`O_EXCL`标志）。有时还可能不太正式地用来指示某些正则表达式修饰符。

### floating point (浮点)

采用“科学计数法”存储数据的一种方法，使数字的精度不依赖于其大小（小数点是“浮动的”）。Perl无法使用整数完成工作时会使用浮点数完成数值计算。浮点数就是实数的近似。

### flush (刷新输出)

清空一个缓冲区的行为，通常在缓冲区满之前。

### FMTEYEWTK

远比你想知道的还要多得多 (Far More Than Everything You Ever Wanted To Know)。对某个小问题的深入研究，如某个超级FAQ问题。更多内容参见Tom。

### foldcase

不考虑大小写完成比较或匹配时Unicode使用的大小写映射。比较小写、标题形式或大写都不可靠，因为Unicode提供了复杂的一对多大小写映射。`foldcase`是专门为解决这个问题而创建的一个小写变体（对某些码点使用一种部分分解规范化形式）。

### fork (派生子进程)

创建在概念上与父进程相等的一个子进程，至少在它有了自己的想法之前。线程会有保护的内存。

### formal arguments (形参)

子例程参数的通用名。在很多语言中，形参总会给定单独的名字；在Perl中，形参就是一个数组中的元素。Perl程序的形参是`$ARGV[0]`、`$ARGV[1]`等。类似地，Perl子例程的形参为`$_[0]`、`$_[1]`等。可以为一个my列表赋值，从而为参数指定不同的名字。另外参见实参。

### format (格式)

这是一个规范，指定在某个地方要放多少个空格、数字等，从而可以得到美观的打印效果。

### freely available (免费可获得)

表示你不用付费就可以得到，不过它的版权仍属于别人（比如Larry）。

### freely redistributable (免费可再发行)

表示如果你私自给朋友一个副本被我们发现，你也不会有法律上的麻烦。实际上，我们宁愿你给所有朋友一个副本。

### freeware (免费软件)

以往，这是指你提供的所有软件，特别是同时提供的软件源代码。现在通常称为开源软件。最近有一个趋势是用这个词来区别开源软件，只表示在免费软件基金会的GPL (General Public License) 下发布的免费软件，不过很难将二者从词源上区别开。

### function (函数)

在数学中，函数就是一个输入值集合映射到一个特定的输出值。在计算机中，这是指返回一个值的子例程或操作符。它可能有输入值（称为参数），也可能没有。

### funny character (趣味字符)

这是指有趣的家伙，就像Larry或者他的某个怪朋友。也指一些奇怪的前缀，Perl要



求将这些趣味字符作为名词标志加到变量上。

### garbage collection (垃圾回收)

这个特性的名字有点不合适，它应该叫做“希望你妈妈帮你收拾屋子”。严格地讲，Perl并不这么做，而是依赖于一种引用计数机制来保证整洁。不过，我们很少严格地讲，通常会把这种引用计数机制认为是垃圾回收的一种形式（值得安慰的是，解释器退出时，如果你因为循环引用之类的问题弄得一塌糊涂，会运行一个“真正的”垃圾回收器来确保把一切都清理好）。

### GID

组ID。在Unix中，这是一个数字组ID，操作系统用来标识你和你的组中的成员。

### glob

严格地讲，就是shell的\*字符，想要生成一个文件名列表时，这会匹配一个字符“团”。不严格地讲，这是指使用glob和类似符号完成模式匹配的行为。另外参见文件团和类型团。

### global (全局)

你在任何地方都能看到的東西，通常是指变量和子例程在程序中的任何地方都可见。在Perl中，只有一些特殊的变量是真正全局的变量，大多数变量（和所有子例程）都只存在于当前包中。全局变量可以用our声明。另外参见第4章中的“全局声明”一节。

### global destruction (全局析构)

这是指Perl解释器关闭时全局变量的垃圾回收（和运行所有相关的对象析构函数）。全局析构并不是大动乱，除非要求这样。

### glue language (粘合语言)

类似Perl的一种语言，可以很好地把本来不打算在一起的东西粘在一起。

### granularity (粒度)

从精神层面讲，所处理的部分的大小。

### grapheme (字形)

石墨(graphene)是一种结晶形碳。六方晶系。作为字形或者更完整的字形簇字符串来讲，这是一个用户可见的字符，它可能包括多个字符(码点)。例如，一个回车加一个换行就是一个字形，但有两个字符，而“ö”是一个字形，但有一个、两

个甚至三个字符，这取决于采用的规范化形式。

### greedy (贪婪)

一种子模式，其量词希望匹配尽可能多的东西。

### grep

原先来自古老的Unix编辑器命令，表示“全局搜索一个正则表达式并打印”，现在用于一般意义上的搜索，特别是文本搜索。Perl有一个内置的grep函数，会搜索与给定规则匹配的一个元素列表，而grep(1)程序会搜索一个或多个文件中与一个正则表达式匹配的行。

### group (组)

你所在的一个用户组。在一些操作系统中（如Unix），可以为你的组中的其他成员提供某些文件访问权限。

### GV

一个内部“glob值”类型定义，包含一个类型团。GV类型是SV的一个子类。

### hacker (黑客)

一些执着地解决技术问题的人，这包括打高尔夫、与半兽人作战或是编程。从道义上讲，黑客是一个中性词。不要把好的黑客与恶意的黑客或刚学编程的愚蠢的小家伙混为一谈。如果你认为他们都一样，那我们会认为你也是有恶意的，或者是愚蠢的。

### handler (处理器)

你的程序需要对某个内部事件做出响应时，如一个信号，或者遇到一个已重载的操作符，Perl会调用这样一个子例程或方法。另外参见回调。

### hard reference (硬引用)

这是一个标量值，包含一个指示对象的具体地址，指示对象的引用计数会把它计算在内（有些硬引用会在内部保存，如从某个类型团变量槽到其相应指示对象的隐式引用）。硬引用与符号引用不同。

### hash (散列)

一个键/值对的无序集合，以适当的方式存储，从而能用一个字符串键很容易地查找其相关的数据值。这个术语表就像一个散列，单词定义为键，定义则是值。散列有时也被称为“关联数据”，因为拼写“关联数据”要麻烦得多，所以大多会简单地



把它称为散列。

### hash table (散列表)

Perl在内部使用的一个数据结构，用来高效地实现关联数组（散列）。另外参见桶。

### header file (头文件)

这个文件中包含某些必要的定义，必须在程序的最前面包含这些定义才能完成某些复杂的操作。C头文件有一个.h扩展名。Perl并没有头文件，不过历史上Perl有时使用了转换的.h文件，但有一个.ph扩展名。另外参见第27章的require（头文件已经被模块机制所取代）。

### here document (here文档)

之所以叫做here文档，是因为shell中有一个类似的构造，它假设命令后面的行（直到某个终止字符串）是提供给这个命令的一个单独的文件。不过，在Perl中，这只是一种有趣的引用形式。

### hexadecimal (十六进制)

一个十六进制数，简称为“hex”。10到16的数字通常表示为字母a到f。Perl中的十六进制常量以0x开头。另外参见第27章的hex函数。

### home directory (主目录)

登录时进入的目录。在一个Unix系统上，这个名字通常由login放在\$ENV{HOME}或\$ENV{LOGDIR}中，不过也可以用(getpwuid(\$<))[7]得到这个目录。（有些平台没有主目录的概念）。

### host (主机)

一个程序或其他数据所在的计算机。

### hubris (傲慢)

过于自负，就像你是神的儿子一样。另外也是一种特点，表示你会编写（和维护）其他人不会说不好的程序。因此，这成为程序员的第三大优秀品质。另外参见懒惰和急躁。

### HV

“散列值”类型定义（hash value）的简写，其中包含一个散列的Perl内部表示。HV类型是SV的一个子类。

### identifier (标识符)

一个合法的名字，这大多是一个计算机程序可能感兴趣的东西。很多语言（包括Perl）允许标识符以一个字母字符开头，然

后可以包含字母和数字。Perl还允许在可以放字母的地方放置连接符号（如下划线字符）。Perl还有更复杂的名字，如限定名。

### impatience (急躁)

计算机表现很懒惰时你感觉到的愤怒。这会使你写出的程序不仅能响应你的需要，实际上还期待更多需求，或者至少假装期望。因此这是程序的第二大优秀品质。另外参见懒惰和傲慢。

### implementation (实现)

一段代码具体如何完成它的工作。代码的用户不能指望实现细节永远都一样，除非它们是发布的接口。

### import (导入)

允许访问另一个模块导出的符号。另外参见第27章中的use。

### increment (增1)

将某个变量的值增1（或者如果指定了另外一个数，则增加这个数）。

### indexing (索引)

原先，这是在一个具体索引（如电话簿）中查找一个键。不过如今这只是使用某种键或位置查找相应的值，即使没有涉及索引。如今已经退化到Perl的index函数只定位一个字符串在另一个字符串中的位置（索引）。

### indirect filehandle (间接文件句柄)

这是一个表达式，计算得到一个可以用作文件句柄的结果，可以是一个字符串（文件句柄名）、一个类型团、类型团引用或底层IO对象。

### indirection (间接性)

如果程序中的某个东西不是你要找的值，而是指示了那个值的位置，这就是间接性。这可以通过符号引用或硬引用来做到。

### indirect object (间接对象)

在英语语法中，动词和它的直接宾语之间的一个名词短语指示了这个动作的接受者。在Perl中，print STDOUT "\$foo\n";可以理解为“动词 间接宾语 宾语”，这里STDOUT是print动作的接受者，“\$foo”是所打印的对象。类似地，调用一个方法时，可以把调用者放在方法和其参数之间的对象槽上：

```
$gollum = new Pathetic::Creature "Sméagol";  
give $gollum "Fisssssh!";  
give $gollum "Precious!";
```

### indirect object slot (间接对象槽)

使用间接对象调用语法时，其语法位置在方法和其参数之间（这个槽由它与下一个参数之间的一个逗号来标识）。在下面的语句中，STDERR就在间接对象槽中：

```
print STDERR "Awake!Awake!Fear, Fire,  
Foes!Awake!\n";
```

### infix (中缀)

操作符放在其操作数之间，如24 \* 7中的乘法操作符(\*)。

### inheritance (继承)

从祖先那里继承的东西，可能是基因方面，也可能是其他方面。如果你正好是一个类，你的祖先就称为基类，你的后代则称为派生类。另外参见单重继承和多重继承。

### instance (实例)

“类的一个实例”的简写，表示该类的一个对象。

### instance data (实例数据)

参见实例变量。

### instance method (实例方法)

对象的一个方法，而不是类方法。

方法的调用者是一个对象，而不是一个包名。类的所有对象都共享该类的全部方法，所以一个实例方法会应用到该类的所有实例，而不是只应用到一个特定的实例。另外参见类方法。

### instance variable (实例变量)

一个对象的属性，随特定对象存储的数据，而不是类作为一个整体存储的数据。

### integer (整数)

没有分数部分的一个数。这是一个可数数，如1, 2, 3等，不过包括0和负数。

### interface (接口)

一段代码承诺永远要提供的服务，不同于其实现，实现可以任意修改。

### interpolation (内插)

将一个标量或列表值插入到另一个值中间，使它看上去好像一直在那里一样。在Perl中，变量内插发生在双引号字符串中，

模式和列表内插出现在构造值列表时，来传入一个列表操作符或者取一个LIST参数的其他类似构造。

### interpreter (解释器)

严格地讲，这是一个读取另一个程序的程序，它直接完成第二个程序所要求的工作，但不会把这个程序先转换为一种不同的形式，那是编译器要做的事情。根据定义，Perl不是一个解释器，因为它包含一种编译器，可以取一个程序，并在perl进程本身中把它转换为一种更可执行的形式（语法树），然后Perl运行时系统再来解释这个语法树。

### invocant (调用者)

要代表这个代理调用一个方法。在一个类方法中，调用者是一个包名。在一个实例方法中，调用者是一个对象引用。

### invocation (调用)

调用一个守护进程、程序、方法、子例程或函数来做你认为它应该做的事情，这个行为就叫做调用。我们通常会调用(call)子例程，但调用(invoke)方法，因为这样听起来更酷一些。

### I/O

对一个文件或设备的输入或输出。

### IO

一个内部I/O对象。也表示间接对象。

### I/O layer (I/O层)

在数据和你得到的输入之间或者数据与最终的输出之间的某个过滤器。

### IPA

印度淡啤酒(India Pale Ale)。也表示国际音标，即全世界用来表示语音记法的标准字母。主要利用Unicode，包括很多组合字符。

### IP

互联网协议(Internet Protocol)，或知识产权(Intellectual Property)。

### IPC

进程间通信(Interprocess Communication)。

### is-a

这是两个对象之间的一种关系，其中一个对象认为是另一个对象的一个更特定的版本，另一个则认为是一个通用对象：“骆



驼是一个哺乳动物”。由于这个通用对象实际上只存在于精神层面，所以我们总是对对象的概念增加一点抽象，把这种关系认为是一个通用基类和一个特定派生类之间的关系。奇怪的是，理想主义的类并不总有理想主义的关系，参见继承。

#### iteration (迭代)

反复地做某件事情。

#### iterator (迭代器)

一种特殊的编程技术，在迭代处理过程中跟踪所在的位置。Perl中的foreach循环包含一个迭代器，散列也有迭代器，允许你调用each分别处理。

#### IV

整数4，不要与6弄混了。这是Tom最喜欢的版本。IV也表示标量类型可以保存的一个内部值 (Integer Value)，不要与NV混淆。

#### JAPH

“Just Another Perl Hacker”，一个聪明但有些神秘的Perl代码，执行时会计算为这个字符串。通常用来展示一个特定的Perl特性，以及USENET签名中见到的Perl大赛里的某些代码。

#### key (键)

一个散列的字符串索引，用来查找与这个键关联的值。

#### keyword (关键字)

参见保留字。

#### label (标签)

为语句指定的名字，这样你就能在程序的任意位置提到这个语句。

#### laziness (懒惰)

这种品质是指你努力减少你总的付出。这会使你编写节省精力的程序，其他人会发现你的程序很有用，然后对你写的程序建立文档，使你不用回答太多有关它的问题。因此，这是程序员的首要品质。当然这也是写这本书的目的。另外参见急躁和傲慢。

#### leftmost longest (左端最长)

正则表达式引擎的首选项，它会匹配一个模式的最左出现，然后给定会出现匹配的位置，这个首选项再查找最长匹配（假设使用了一个贪婪量词）。另外参见第5章来

了解有关这个主题的更多内容。

#### left shift (左移)

这种移位会将数字乘以2的一个幂。

#### lexeme (词法单位)

token的一种有趣叫法。

#### lexer (词法分析器)

词法分析工具 (tokenizer) 的一种有趣叫法。

#### lexical analysis (词法分析)

词法分析 (tokenizing) 的一种有趣叫法。

#### lexical scoping (词法作用域)

用显微镜查看你的牛津英语大辞典（也称为静态作用域，因为词典不会太快改变）。类似地，查看存储在各个作用域的一个私有字典（命名空间）中的变量，这些变量只是从其声明到其所在词法作用域末尾的范围内可见。同义词：静态作用域，反义词：动态作用域。

#### lexical variable (词法作用域变量)

有词法作用域的变量，由my声明。通常称为一个“lexical”（our为一个全局变量声明一个词法作用域名，它本身并不是词法作用域变量）。

#### library (库)

一般地，这是一个过程的集合。原先，这是指一个.pl文件中的子例程集合。现在，更常用来指示你的系统上的整个Perl模块集合。

#### LIFO

后进先出 (Last In, First Out)。另外参见FIFO。LIFO通常称为一个堆栈。

#### line (行)

在Unix中，0个或多个非换行符字符的序列，以一个换行符结束。在非Unix机器上，这由C库模拟，尽管底层操作系统有不同的想法。

#### linebreak (换行)

这是一个字形，由一个回车加一个换行（或者任何有Unicode Vertical Space字符属性的字符）组成。

#### line buffering (行缓冲)

标准I/O输出流使用行缓冲，在每个换行符后刷新输出其缓冲区。很多标准I/O库自动为将发送到终端的输出建立行缓冲。



### line number (行号)

这一行之前读入的行数加1。Perl为它打开的每个源或输入文件维护了一个单独的行号。当前源文件的行号由`__LINE__`表示。当前输入行号(最近通过`<FH>`读取的文件)由`$($INPUT_LINE_NUMBER)`变量表示。很多错误消息中这两个值都会报告(如果有)。

### link (链接)

作为名词使用时,这是目录中的一个名字,表示一个文件。一个给定的文件可以有多个链接。这就像电话目录中同一个电话号码可能列在不同的名字下面。作为一个动词,这是指将一个部分编译文件的未解析符号解析为一个(接近)可执行的映像。链接可以是静态或动态的,这与静态或动态作用域没有任何关系。

### LIST

这是一个语法构造,表示一个用逗号分隔的表达式列表,可以计算得到一个列表值。`LIST`中的每个表达式将在列表上下文中计算,并内插到这个列表值中。

### list (列表)

标量值的一个有序集合。

### list context (列表上下文)

在这种情况下,表达式周围的环境(调用它的代码)希望表达式返回一个值列表而不是一个值。希望得到一个参数列表的函数会告诉这些参数它们都要生成一个列表值。另外参见上下文。

### list operator (列表操作符)

处理列表值的一个操作符,如`join`或`grep`。通常用于命名内置操作符(如`print`、`unlink`和`system`),它们不需要在其参数列表两边加小括号。

### list value (列表值)

临时标量值的一个匿名列表,可以在程序中从任何生成列表的函数传入提供一个列表上下文的函数或构造。

### literal (直接量)

编程语言中的一个token,如一个数字或字符串,这些token提供实际的值,而不像变量一样只是表示可能的值。

### little-endian (小端)

源自斯威夫特的《格列佛游记》:有人吃鸡蛋时先吃小头一端。也指一些计算机把

一个字的低位字节存储在低字节地址而不是高位字节。通常认为优于大端机器。另外参见大端。

### local (局部)

不表示所有地方都一样。Perl中的全局变量可以通过`local`操作符在一个动态作用域中局部化。

### logical operator (逻辑操作符)

表示“与”、“或”、“异或”和“非”等概念的符号。

### lookahead (前瞻)

查看当前匹配位置右边字符串的断言。

### lookbehind (后顾)

查看当前匹配位置左边字符串的断言。

### loop (循环)

这个构造会重复做某件事。就像一个过山车。

### loop control statement (循环控制语句)

循环体中可以使循环永久停止循环或跳过一次迭代的语句。一般地,不要在过山车上使用这种构造。

### loop label (循环标签)

与一个循环(或过山车)关联的一种键或名字,使循环控制语句可以指定它们想要控制哪个循环。

### lowercase (小写)

在Unicode中,不只是通用类别为Lowercase Letter的字符,而是有Lowercase属性的所有字符,包括修饰符字母、字母数字、其他符号和一个组合标记。

### lvaluable (可作为左值)

可以作为一个左值。

### lvalue (左值)

语言学家使用的专业术语,表示一个存储位置,你可以向这个位置赋一个新值,如变量或一个数组的一个元素。“l”是“左”(left)的简写,表示赋值的左边,这是左值的一个常见位置。可作为左值的函数或表达式是可以赋值的函数或表达式,如`pos($x)=10`。

### lvalue modifier (左值修饰符)

这是一个形容词性的伪函数,以某种声明性方式调整左值的含义。目前有3个左值修饰符: `my`, `our`和`local`。

### magic (魔法)

技术上讲, 与一个变量(如`$!`, `$0`, `%ENV`或`%SIG`)或任何绑定变量关联的额外语义。调整这些变量时就会有魔法发生。

### magical increment (魔法自增)

这个自增操作符除了可以让数字增加外, 还知道如何增加ASCII字母。

### magical variables (魔法变量)

访问这些特殊变量或对它们赋值时会有一些副作用。例如, 在Perl中, 修改`%ENV`数组的元素还会改变将要使用的相应环境变量。读取`$!`变量会提供当前系统错误号或消息。

### Makefile

这个文件控制一个程序的编译。Perl程序通常不需要一个Makefile, 因为Perl编译器有充分的自我控制功能。

### man

这个Unix程序会为你显示在线文档(手册页)。

### manpage

手册中的一“页”, 通常通过`man(1)`命令访问。手册页包含SYNOPSIS、DESCRIPTION、BUGS列表等等, 通常比一页长。对于命令、系统调用、库函数、设备、协议、文件等都有一些手册页提供文档。这本书中, 我们把任何标准Perl文档(如`perlop`或`perldelta`)都称为手册页, 不论你的系统上以什么格式安装。

### matching (匹配)

参见模式匹配。

### member data (成员数据)

参见实例变量。

### memory (内存)

通常表示你的主存, 而不是磁盘。麻烦的是你的机器可能实现了虚拟内存, 也就是说, 它会假装有比实际更多的内存, 而且会使用磁盘空间保存不活动的数据。这可能看起来像是你的内存比实际更多一点, 不过这不能取代实际内存。关于虚拟内存, 最好的一点是它允许性能逐步下降, 而不是突然耗尽全部实际内存。不过只要你还没有先整垮你的磁盘, 程序可能会在耗尽虚拟内存时退出。

### metacharacter (元字符)

不会以正常方式处理的字符。哪些字符要

作为元字符而需要特殊处理, 这会随上下文不同而有很大变化。你的shell可能有一些元字符, 双引号Perl字符串可能有另外一些元字符, 而正则表达式模式会有所有双引号元字符以及另外一些自己的元字符。

### metasymbol (元符号)

这些可以称为元字符, 只不过这是字符序列而不是一个字符。一般地, 这个序列中第一个字符必须是一个真正的元字符, 使得元符号中的其他字符有与平常不同的表现。

### method (方法)

对象可以采取的一种动作(如果你告诉对象这么做)。另外参见第12章。

### method resolution order (方法解析顺序)

Perl从`@INC`得到的路径。默认地, 这是一个双重深度优先搜索, 一次查找定义的方法, 另一次查找AUTOLOAD。不过, Perl允许你用`mro`来配置。

### minicpan

这是一个CPAN镜像, 只包含各个发行版的最新版本, 可能用`CPAN::Mini`创建。另外参见第19章。

### minimalism (最小主义)

坚信“小才美丽”。讽刺的是, 如果用小语言讲话, 会说很多话, 而如果用一个大语言来讲, 只会讲很少。可以试试看。

### mode (模式)

在`stat(2)`系统调用上下文中, 这是指包含权限位和文件类型的字段。

### modifier (修饰符)

参见语句修饰符、正则表达式修饰符和左值修饰符, 不一定严格按照这个顺序。

### module (模块)

定义(几乎)与模块同名的包的文件, 可以导出符号或函数作为对象类(模块的主`.pm`文件还可以加载其他文件来支持模块)。另外参见`use`内置函数。

### modulus (取模)

这是一个整数除法运算, 你只对余数感兴趣而不关心商。

### mojibake (乱码)

你讲一种语言, 而计算机以为你在讲另一种语言。例如, 你在发送UTF-8, 不过计



算机认为你在发送Latin-1，你会看到奇怪的转换，显示各种奇怪的字符。用日语写就是文字化行，这表示“字符腐烂”，这个描述很贴切。在标准IPA音标里读作[modzibake]，或者大致为“moh-jee-bah-keh”。

#### monger

*Perlmongers*成员的简写，这是Perl的提供者。

#### mortal (将死亡)

当前语句完成时将要消亡的一个临时值。

#### mro

参见方法解析顺序。

#### multidimensional array (多维数组)

用多个下标来查找一个元素的数组。Perl使用引用来实现多维数组。另外参见第9章。

#### multiple inheritance (多重继承)

你从父母那里遗传得到的特性，会以不可预测的方式混合（参见继承和单重继承）。在计算机语言中（包括Perl），这个概念是指一个给定的类有多个直接祖先或基类。

#### named pipe (命名管道)

文件系统中一个有名字的管道，可以由两个无关的进程访问。

#### namespace (命名空间)

名字域。不用担心这个域中的这些名字是否在其他域中使用。另外参见包。

#### NaN

不是一个数。对于某些不合法或不能表述的浮点操作，Perl会使用这个值。

#### network address (网络地址)

套接字最重要的属性，就像你的电话号码。通常是一个IP地址。另外参见端口。

#### newline (换行符)

表示行末的一个字符，Unix下ASCII值为八进制012（不过在Mac上是015），在Perl字符串中用\n表示。对于写文本文件的Windows机器以及一些物理设备（如终端），这个换行符会由你的C库自动转换为一个换行加一个回车，不过正常情况下不会完成转换。

#### NFS

网络文件系统，允许你装载一个远程文件

系统，就好像它是本地文件系统一样。

#### normalization (规范化)

将一个文本字符串转换为一个候选但等价的标准（或兼容的）表示，可以用来比较相等性。Unicode识别4种不同的规范化形式：NFD、NFC、NFKD和NFKC。

#### null character (null字符)

数值为0的字符。C用这个字符来结束字符串，不过Perl允许字符串包含null。

#### null list (null列表)

有0个元素的列表值，在Perl中用()表示。

#### null string (null串)

包含0个字符的字符串，不要与包含一个null字符的字符串混淆，后者的长度大于0，而且为true。

#### numeric context (数值上下文)

在这种情况下，一个表达式周围的代码（调用它的代码）希望它返回一个数字。另外参见上下文和字符串上下文。

#### numification (数值化)

有时读作*nummification*和*nummify*。这是Perl的一个术语，表示隐式转换为一个数，相关的动词是*numify*（数字化）。Numification（数值化）是为了与*mummification*压韵，*numify*与*mummify*压韵。这与英语里的*numen*、*numina*、*numinous*没有任何关系。我们很早以前忘记了额外的m，一些人已经习惯这种有趣的拼法，另外HTTP\_REFERER自己也漏了字母，所以我们这种奇怪的拼法得以保留。

#### NV

内华达州（Nevada）的简写，不要与文明混淆。NV还表示一个标量可以保存的内部浮点数值类型，不要与IV混淆。

#### nybble (半字节)

半个字节，等价于一个十六进制位，相当于4个二进制位。

#### object (对象)

类的一个实例。它“知道”自己是什么用户自定义的类型（类），以及作为这个类可以做哪些事情。你的程序可以请求一个对象来完成工作，不过对象会自己决定它是否想做这个工作。有些对象会更随和一些。

#### octal (八进制)



一个八进制数。只允许0到7范围内的数。Perl中的八进制常量从0开始，如013。另外参见oct函数。

#### offset (偏移量)

从字符串或数组开头移动时，要跳过多少个元素才能到达其中一个指定的位置，因此，最小偏移量是0，而不是1，因为你不用跳过任何元素就可以到达第一个元素。

#### one-liner (单行代码)

整个计算机程序只有一行文本。

#### open source software (开源软件)

这些程序的源代码可以免费得到，免费再发行，并没有任何商业要求。要了解更详细的定义，参见<http://www.opensource.org/osd.html>。

#### operand (操作数)

得到一个值的表达式，将由操作符处理。另外参见优先级。

#### operating system (操作系统)

一个特殊的程序，在裸机上运行，会隐藏管理进程和设备的复杂细节。通常不太严格地用于指示一个特定的编程文化。可以在各个不同层次的特定性上放宽限制。在一个极端上，可以把Unix和类Unix的所有版本都认为是同一个操作系统（这会让很多人不高兴，特别是律师和其他拥护者）。在另一个极端，可以说这个开发者的这个特定版本的操作系统不同于它的任何其他版本，或者任何其他开发者的操作系统。与其他很多语言相比，Perl在不同操作系统上有更好的可移植性。另外参见体系结构和平台。

#### operator (操作符)

一个将一些数值转换为一些输出值的小玩艺，通常是语言内置的，有特定的语法或符号。给定的操作符对于参数（操作数）接收的数据类型以及从这个操作符返回的数据类型会有特定的期望。

#### operator overloading (操作符重载)

这是一种可以在内置操作符上完成的重载，从而可以作用于对象，就好像对象是普通的标量值一样，不过有对象类提供的具体语义。这是用overload pragma建立的，参见第13章。

#### options (选项)

参见开关或正则表达式修饰符。

#### ordinal (序数)

一个抽象字符的整数值，与码点含义相同。

#### overloading (重载)

为一个符号或构造给定另外的含义。实际上，所有语言都会以某种程度完成重载，因为人们很擅长根据上下文做出区别。

#### overriding (覆盖)

隐藏同一个名字的其他定义，或置其无效（不要与重载混淆，重载是增加额外的定义，必须以另外某种方式来区别）。更容易让人混淆的是，我们使用这个词时有两个重载的定义：一方面描述如何定义你自己的子例程来隐藏一个同名的内置函数（参见第11章“覆盖内置函数”一节），另外还用于描述如何在一个派生类中定义一个替换方法来隐藏同名的基类方法（参见第12章）。

#### owner (所有者)

对一个文件有绝对控制权的一个用户（除超级用户以外）。文件还可以有一个用户组，如果实际用户允许的话，这些用户可以有联合所有权。另外参见权限位。

#### package (包)

全局变量、子例程等等的命名空间，这样它们可以与不同命名空间中有类似名字的符号相区分。从某种意义上讲，只有包是全局的，因为包符号表中的符号要从包外编写的代码访问（必须指定这个包名）。不过，在另一个方面，所有包符号也是全局的，它们都是得到妥善组织的全局变量。

#### pad (便签)

便签簿 (scratchpad) 的简写。

#### parameter (参数)

参见参数。

#### parent class (父类)

参见基类。

#### parse tree (解析树)

参见语法树。

#### parsing (解析)

这是一个很琐碎但有时也很野蛮的技术，要将可能不合法的程序转换为一个合法的语法树。

### patch (补丁)

可以说是加一个东西来修补。在黑客王国里，你想要修正一个bug或升级你的老版本时，*patch(1)*程序可以提供程序的两个版本之间的差别列表。

### PATH

系统查找你想要执行的程序时所搜索的目录列表。这个列表存储为一个环境变量，在Perl中可以通过\$ENV{PATH}访问。

### pathname (路径名)

一个完全限定文件名，如*/usr/bin/perl*。有时可能与PATH混淆。

### pattern (模式)

模式匹配中使用的一个模板。

### pattern matching (模式匹配)

取一个模式，通常是一个正则表达式，在一个字符串上以不同方式尝试这个模式，查看是否有办法让它们匹配。通常用于从文件中选择有趣的东西。

### PAUSE

The Perl Authors Upload SErver (*http://pause.perl.org*)，这是模块进入CPAN的必经大门。

### Perl mongers

一个Perl用户组，名字由New York Perl mongers得来（这是第一个Perl用户组）。可以在*http://www.pm.org*寻找你附近的这样一个用户组。

### permission bits (权限位)

文件的所有者设置或取消设置的一些位，从而允许或不允许其他人访问。这些标志位是询问文件情况时stat内置函数返回的模式字的一部分。在Unix系统上，可以检查*ls(1)*手册页来了解更多信息。

### Pern

完成两次Perl++可以得到。如果只做一次，那只能卷发(perm)。必须加8次才能洗好头发。先揉出泡沫，然后冲洗，再重复这个过程。

### pipe (管道)

一个直接连接，将一个进程的输出传送到另一个进程的输入，而没有任何中间的临时文件。一旦建立了管道，当前的两个进程可以顺利读写，就好像它们在与一个正常的文件通信一样，不过有一些注意事

项。

### pipeline (管线)

连续的一系列进程，由管道连接，每一个进程将其输出流传递到下一个进程。

### platform (平台)

程序运行所在的整个硬件和软件环境。如果改变了以下某个方面：机器、操作系统、库、编译器或系统配置，用一种依赖于平台的语言编写的程序可能无法正确工作。对于各个平台，Perl解释器必须以不同方式编译，因为它用C实现的，不过用Perl语言编写的程序大多是平台无关的。

### pod

这种标记用来在你的Perl代码中嵌入文档。Pod代表“Plain old documentation”。另外参见第23章。

### pod command (pod命令)

这是一个序列，如=head1，这指示一个pod节的开始。

### pointer (指针)

像C之类的语言中的一个变量，其中包含另外某一项的具体内存位置。Perl在内部处理指针，所以你不用担心这个方面。可以使用键和变量名或硬引用形式使用符号指针，它们不是指针（不过像指针，而且确实包含了指针）。

### polymorphism (多态)

这个概念是指你可以告诉一个对象完成某个通用处理，这个对象会根据其类型以不同的方式解释这个命令。“< 源于希腊语 πολυ- + μορφή，表示很多形态”。

### port (端口)

一个TCP或UDP套接字地址的一部分，找到正确的机器后将数据包定向到正确的进程，有点像给公司接线员打电话时告诉她的分机号。另外还有移植的意思，这是转换代码，从而在不同于原先计划的平台上运行的结果，或者是指示这种转换的动词。

### portable (可移植)

以前，C代码可以在BSD和SysV上编译。一般地，可移植的代码可以很容易地转换，从而在另一个平台上运行，这里“很容易”可以有不同的定义。如果你足够努力，任何代码都可以认为是可移植的，如移动房屋和伦敦大桥。



### porter (移植人员)

将软件从一个平台“转移”到另一个平台的人。对于用依赖于平台的语言(如C)编写的程序,移植可能很费劲,而移植Perl程序会相当容易。

### possessive (占有的)

是指模式中的量词和组一旦占有就拒绝放弃。更正式的说法是不可回溯,不过“占有”更简洁。

### POSIX

可移植操作系统接口规范(The Portable Operating System Interface specification)。

### postfix (后缀)

位于操作数后面的操作符,如`$x++`。

### pp

“压入-弹出”(push-pop)代码的内部简写。也就是说,实现Perl堆栈机的C代码。

### pragma

一个标准模块,在编译时接收(或忽略)它的实用提示和建议。pragma名字应当是全小写。

### precedence (优先级)

如果没有其他规则,要按照这个优先级顺序确定哪个先发生,例如,如果没有括号,总是先乘除再加减。

### prefix (前缀)

放在操作数前面的操作符,如`++$x`。

### preprocessing (预处理)

一些辅助进程完成的工作,将到来的数据转换为更适合当前进程的一种形式。通常用一个管道完成。参见C预处理器。

### primary maintainer (主要维护者)

模块作者,PAUSE允许他为一个命名空间指定合作维护者权限。主要维护者可以放弃这个特权,把它转交给另一个PAUSE作者。另外参见第19章。

### procedure (过程)

一个子例程。

### process (进程)

运行的程序的一个实例。在多任务系统中,如Unix,两个或多个不同的进程可以同时独立地运行同一个程序。实际上,fork函数就是设计来提供这种服务的。

在其他操作系统中,进程有时称为“线程”、“任务”或“作业”,通常含义上有细小的差别。

### program (程序)

参见脚本。

### program generator (程序生成器)

采用一种高级语言利用算法为你写代码的系统。另外参见代码生成器。

### progressive matching (渐进匹配)

从之前离开的位置继续模式匹配。

### property (属性)

参见实例变量或字符属性。

### protocol (协议)

在网络中,双方协商好的一种来回发送消息的方式,使双方都不会混淆。

### prototype (原型)

子例程声明中可选的一部分,告诉Perl编译器要传递多少个参数作为实参(以及参数是什么类型),使你能够编写像内置函数一样解析的子例程调用(或者很可能不解析)。

### pseudofunction (伪函数)

这个构造有时看上去像函数,但实际上不是。通常为一些修饰符保留,如左值修饰符(如my),上下文修饰(如scalar),以及自选引号构造(q//, qq//, qx//, qw//, qr//, m//, s///, y///和tr///)。

### pseudohash (伪散列)

原先,这是一个数组引用,该数组中初始的元素正好包含一个散列的引用。以前可以把一个伪散列引用处理为数组引用或散列引用。现在已经不再支持伪散列。

### pseudoliteral (伪直接量)

看上去像直接量的操作符,如输出获取操作符``command``。

### public domain (公共域)

不属于任何人专有。Perl有版权,所以不在公共域中。不过它可以免费得到,免费再发行。

### pumpkin

这是一个“接力棒”概念,在Perl社区传递,指示谁在某个开发领域是主要集成人员。



## pumpking

*pumpkin*所有者，这个人负责推动开发，或者至少摆出这种姿态。这个人必须乐于不时担任Great Pumpkin角色。

## PV

“指针值” (pointer value)，这是Perl对char\*的内部说法。

## qualified (限定)

拥有一个完整的名字。符号\$Ent::moot是限定的，\$moot则不是限定的。一个完全限定的文件名从顶层目录指定。

## quantifier (量词)

正则表达式的一个组件，指示后面的原子出现多少次。

## race condition (竞态条件)

如果多个相互关联的事件的结果依赖于这些事件的顺序，但是这个顺序由于非确定性的计时效果而无法保证，这就会存在竞态条件。如果两个或多个程序或同一个程序的不同部分试图完成同样的事件序列，一个可能中断另一个的工作。这是一种发现漏洞的好办法。

## readable (可读)

对于文件，可能已经设置了适当的权限位置来允许你访问这个文件。对于计算机程序，这表示编写得当，使人能确定它要做什么。

## reaping (俘获)

父进程代表死亡的子进程完成的最后仪式，使它不再是一个僵尸。另外参见wait和waitpid函数调用。

## record (记录)

文件或流中一组相关的数据值，通常与一个唯一的键字段关联。在Unix中，通常等同于一行，或以空行终止的一个行集（或段落）。/etc/passwd文件的每一行是一个记录，以登录名为键，包含有关这个用户的信息。

## recursion (递归)

用自己来定义自己（至少部分）的艺术，在字典中这是不可能的，但是在计算机程序中，只要你注意不要陷入无限递归（这就像一个有更壮观失败模式的无限循环），这就是可以的。

## reference (引用)

在这里查找指向其他位置上信息的指针（参见间接）。引用有两种类型：符号引用和硬引用。

## referent (指示对象)

引用指示的对象，这可能有名字，也可能没名字。指示对象的常用类型包括标量、数组、散列和子例程。

## regex (正则表达式)

参见正则表达式。

## regular expression (正则表达式)

一个有多种解释的实体，如大象。对于计算机科学家来说，这是一个小语言的文法，其中一些字符串是合法的，另外一些不合法。对于普通人来说，这是一个模式，可以用来查找不同情况下可能变化的东西。Perl正则表达式的规则性并不体现在理论意义上，而是在于其使用。下面是一个正则表达式：/Oh s.\*t./。这会匹配类似“Oh say can you see by the dawn's early light”和“Oh sit!”的字符串。参见第5章。

## regular expression modifier (正则表达式修饰符)

模式或替换中的一个选项，如/i会使模式不区分大小写。

## regular file (常规文件)

这个文件不是目录、设备、命名管道或套接字，或符号链接。Perl使用-f文件测试操作符来标识常规文件。有时称为“普通”文件。

## relational operator (关系操作符)

这个操作符指示对于一对操作数，某个特定的顺序关系是否为true。Perl提供了数值和字符串关系操作符。另外参见排序序列。

## reserved words (保留字)

对编译器有特定的内置含义的词，如if或删除。在很多语言中（不包括Perl），使用保留字命名其他对象的做法是非法的（毕竟，这也是为什么要保留它们的原因）。在Perl中，只是不能用这些保留字命名标签或文件句柄。也称为“关键字”。

## return value (返回值)

一个子例程或表达式在计算时生成的值。在Perl中，返回值可以是一个列表或一个标量。

## RFC

请求评论 (Request For Comment)，尽管名字比较含蓄，这实际上是一系列重要的标准文档。

## right shift (右移)

这种移位会将数字除以2的一个幂。

## role

一个具体行为集合的名字。角色是不通过继承为一个类增加行为的一种方法。

## root

超级用户 (UID == 0)。也是文件系统的顶级目录 (根)。

## RTFM

有人认为你该读读手册时就会讲RTFM。

## run phase (运行阶段)

Perl开始运行你的主程序之后的阶段。另外参见编译阶段。运行阶段主要发生在运行时，不过执行require、do *FILE*或eval *STRING*操作符时，或者一个替换使用/ee修饰符时，也会发生在编译时。

## runtime (运行时)

此时Perl会具体完成代码所要求的工作，而不同于之前的编译时，在编译时，Perl会尝试确定你的代码是否有意义。

## runtime pattern (运行时模式)

这个模式包含一个或多个变量，在把模式解析为一个正则表达式之前需要内插这些变量，因此不能在编译时分析，而必须在每次计算模式匹配操作符时重新分析。运行时模式很有用，但是开销很大。

## RV

再造自行车 (recreational vehicle)，不要与车辆娱乐 (vehicular recreation) 混淆。RV也表示一个标量可以保存的内部引用值类型。另外参见IV和NV (如果你还没糊涂的话)。

## rvalue (右值)

这是在赋值右边可能看到的值。另外参见左值。

## sandbox (沙箱)

一个被围起来的区域，不会影响围墙外的东西。你会让孩子在沙箱里玩，而不是在马路上玩。参见第20章。

## scalar (标量)

一个简单的单数值；一个数、字符串或引用。

## scalar context (标量上下文)

在这种情况下，表达式周围的环境 (调用它的代码) 希望表达式返回一个值而不是一个列表。另外参见上下文和列表上下文。标量上下文有时会对返回值施加的约束，参见字符串上下文和数值上下文，有时我们会谈到条件中的布尔上下文，不过这没有任何额外的约束，因为任何标量值 (不论是数值还是字符串) 已经为true或false。

## scalar literal (标量直接量)

一个数字或加引号的字符串——这是程序文本中的具体值，与变量不同。

## scalar value (标量值)

值为标量而不是列表。

## scalar variable (标量变量)

有\$前缀的变量，其中包含一个值。

## scope (作用域)

在多大范围内可以看到一个变量。Perl有两种可见性机制。对于局部变量有动态作用域，这表示块的其余部分以及由块其余部分调用的子例程可以看到这个块的局部变量。Perl对my变量了提供词法作用域，表示块的其余部分可以看到这个变量，但是由这个块调用的其他子例程无法看到这个变量。

## scratchpad (便签簿)

一个特定文件或子例程的特定调用会在这个区域保存它的一些临时值，包括所有词法作用域变量。

## script (脚本)

这是一个文本文件，程序想要直接执行，而不会在执行之前先编译为另外一种文件形式。另外在Unicode上下文中，这表示一个特定语言或一组语言的文字系统，如希腊语、孟加拉语或谈格瓦文字。

## script kiddie (脚本小子)

自以为是黑客但实际上不是黑客，只知道一点皮毛，会运行包装好的脚本而已。这是一个有船货崇拜综合症的程序员。

## sed

一个令人尊敬的流编辑器 (Stream Editor)，Perl从中汲取了很多想法。



### semaphore (信号量)

一种有趣的锁，防止多个进程或线程同时使用同一个资源。

### separator (分隔符)

保证周围字符串不会相互混淆的一个字符或字符串。*split*函数要按分隔符分解。不要与定界符或终止符混淆。前面所说的“或”就分隔了两个候选的概念。

### serialization (串行化)

将一个复杂的数据结构转换为线性顺序，从而可以作为一个字符串存储在一个磁盘文件或数据库中，或者通过一个管道发送。也称为编组。

### server (服务器)

在网络中，这个进程发布一个服务，或者留在某个已知位置上等待需要服务的客户端与它联系。

### service (服务)

你为别人做的一件让他们高兴的事情，如告诉他们时间（或者他们的寿命）。在一些机器上，*getservent*函数会列出一些公共已知的服务。

### setgid

等同于*setuid*，只不过是交出组特权。

### setuid

是指一个程序用其所有者权限运行而不是（通常的）运行该程序的人的权限。另外还描述控制特性的模式字（权限位）中的相应位。这个位由所有者显式设置，以启用这个特性，程序必须当心，不要过度交出本来不该交出的特权。

### shared memory (共享内存)

一段两个不同进程都可以访问的内存，否则它们不会看到对方的内存。

### shebang

爱尔兰语，表示整个McGillicuddy。在Perl文化中，这是“sharp”和“bang”的组合，表示#!序列，告诉系统到哪里寻找解释器。

### shell

这是一个命令行解释器。程序交互式地提供一个提示，接收一行或多行输入，并执行你提到的程序，为这些程序分别输入适当的参数和输入数据。Shell还可以执行包含这些命令的脚本。在Unix下，典型的

shell包括Bourne shell (*/bin/sh*)、C shell (*/bin/csh*)和Korn shell (*/bin/ksh*)。Perl严格地不能算是一个shell，因为它不是交互性的（尽管Perl程序可以是交互性的）。

### side effects (副作用)

计算一个表达式时额外发生的事情。如今这可以表示几乎任何事情。例如，计算一个简单的赋值语句通常会有一个“副作用”：一个变量赋值（你可能认为赋值正是你的本来目的）。类似地，为特殊变量\$| (*\$AUTOFLUSH*) 赋一个值会有一个副作用：对当前选择的文件句柄的每个write或print之后都强制一个刷新输出。

### sigil (印记)

一个魔法字形。或者，对于Perl来说，这是放在一个变量名前面的符号，如\$、@和%。

### signal (信号)

晴天霹雳；也就是说，这是由操作系统触发的事件，可能会在你不希望的时候触发。

### signal handler (信号处理器)

一个子例程并不满足于以正常方式调用，它会等待一个晴天霹雳（执行这个子例程就是为了处理这个信号）。在Perl下，晴天霹雳称为信号，可以用kill内置函数发送。另外参见第25章中的%SIG散列和第15章中的“信号”一节。

### single inheritance (单重继承)

你从你母亲那些继承的特性（如果她告诉你没有父亲）。参见继承和多重继承。在计算机语言中，其思想是：类都是无性繁殖，所以一个给定的类只能有一个直接祖先或基类。Perl没有提供这种限制，不过如果你愿意，当然可以这样写Perl程序。

### slice (片段)

一个列表、数组或散列中选择的任意多个元素。

### slurp (吞入)

用一个操作将整个文件读入一个字符串。

### socket (套接字)

多个进程间实现网络通信的一个端点，类似于一个电话或一个邮箱。关于套接字最重要的一点是它的网络地址（类似于电话号码）。不同类型的套接字有不同类型的地址，有些看起来像文件名，有些则不是。



**soft reference (软引用)**

参见符号引用。

**source filter (源过滤器)**

一种特殊的模块，在脚本进入词法分析器之前先由这个模块对脚本进行预处理。

**stack (堆栈)**

利用这个设备，你可以把东西放在最上面，然后按放入时相反的顺序取出东西。另外参见LIFO。

**standard (标准)**

包含在官方Perl发行版本中，如标准模块、标准工具或标准Perl手册页。

**standard error (标准错误)**

不属于标准输出的错误消息的默认输出流。在Perl程序中由文件句柄STDERR表示。可以显式使用这个流，不过die和warn内置函数会自动写至你的标准错误流（除非被捕获或截获）。

**standard input (标准输入)**

程序的默认输入流，如果可能，不用关心它的数据来自哪里。在Perl程序中由文件句柄STDIN表示。

**standard I/O (标准I/O)**

标准C库，用于向操作系统完成缓冲输入和输出（标准I/O中的“标准”最多只是与标准输入和输出中的“标准”有关）。一般地，Perl依赖于一个给定操作系统提供了怎样的标准I/O实现，所以一个机器上某个Perl程序的缓冲特性可能与另一个机器上该程序的缓冲特性并不一致。正常情况下，这只会影响效率，而不影响语义。如果你的标准I/O包会完成块缓冲，而你希望更频繁地刷新输出这个缓冲区，只需要把\$!变量设置为一个true值。

**Standard Library (标准库)**

官方Perl发行版本的所有内容。有些开发者的Perl版本会改变其发行版本，去掉某些部分，或者增加额外的一些部分。另外参见双重性。

**standard output (标准输出)**

程序的默认输出流，如果可能，不用关心它的数据去往哪里。在一个Perl程序中由文件句柄STDOUT表示。

**statement (语句)**

提供给计算机的一个命令，告诉它下一步

做什么，类似于菜谱里的下一个步骤：

“在黄油里加些果酱，然后搅匀。”语句与声明不同，声明不会告诉计算机做任何事情，只是要了解某些事情。

**statement modifier (语句修饰符)**

在语句后面而不是前面放一个条件或循环，你应该知道我们的意思。

**static (静态)**

与其他的東西相比变化很缓慢（遗憾的是，所有一切都相当稳定，除了某些基本粒子，我们对这些基本粒子的特性还不太确定）。在计算机中，在这里所有东西都认为会快速变化，“静态”的概念略有弱化，指示一个功能稍有些故障的变量、子例程或方法。在Perl文化中，已经委婉地避开了这个词。

如果你是一个C或C++程序员，你可能会查找Perl的state关键字。

**static method (静态方法)**

没有这种东西。参见类方法。

**static scoping (静态作用域)**

没有这种东西。参见词法作用域。

**static variable (静态变量)**

没有这种东西。只需要在一个比子例程更大的作用域使用词法作用域变量，或者用state声明而不是my。

**stat structure (stat结构)**

一个特殊的内部位置，Perl在这里保存你请求信息的最后一个文件的有关信息。

**status (状态)**

某个子进程死亡时返回给父进程的值。这个值放在特殊变量\$?中。其高8位是死亡进程的退出状态，低8位标识让这个进程死亡的信号（如果有）。在Unix系统上，这个状态值等于wait(2)返回的状态字。参见第27章的system。

**STDERR**

参见标准错误。

**STDIN**

参见标准输入。

**STDIO**

参见标准I/O。

**STDOUT**

参见标准输出。

### stream (流)

作为一个稳定的字节或字符序列流入或流出进程的一个流，不会分解为数据包。这是一种接口，底层实现可能会把数据分解为单独的数据包来进行传输，不过这会对你隐藏。

### string (字符串)

一个字符序列，如 “He said!@#\*&%@#\*?!”。字符串不一定是完全可打印的。

### string context (字符串上下文)

在这种情况下，表达式周围的环境（调用它的代码）希望表达式返回一个字符串。另外参见上下文和数值上下文。

### stringification (字符串化)

为一个抽象对象生成一个字符串表示的过程。

### struct

引入一个结构定义或名字的C关键字。

### structure (结构)

参见数据结构。

### subclass (子类)

参见派生类。

### subpattern (子模式)

正则表达式模式的一部分。

### subroutine (子例程)

一个命名或可访问的程序段，可以从程序中任何其他位置调用，来完成程序的某个子目标。子例程通常有参数，从而可以根据其输入参数完成不同但相关的事情。如果子例程返回一个有意义的值，它也称为函数。

### subscript (下标)

这个值指示一个特定的数组元素在数组中的位置。

### substitution (替换)

通过 `s///` 操作符改变一个字符串的一部分（我们避免使用这个词来表示变量内插）。

### substring (子串)

一个字符串的一部分，从某个字符位置（偏移量）开始的一定数目的字符。

### superclass (超类)

参见基类。

### superuser (超级用户)

操作系统允许这个人做几乎任何事情。通常是你的系统管理员或者假装是你的系统管理员。在Unix系统上，这就是 *root* 用户。在Windows系统上，通常是管理员用户。

### SV

“标量值” (scalar value) 的简写。不过在Perl解释器中，通过采用一种面向对象的方式，每个指示对象都处理为从SV派生的一个类的成员。Perl中的每个值都作为C语言SV\*指针传递。SV *struct* 知道自己的“指示对象类型”，而且代码足够聪明（我们希望如此），不会尝试在一个子例程上调用一个散列函数。

### switch (开关)

在命令行上指定的选项，会影响你的程序如何工作，通常用一个负号引入。这个词也用作 *switch* 语句的一个昵称。

### switch cluster (形状簇)

多个命令行开关（如 `-a -b -c`）组合为一个开关（如 `-abc`）。有额外参数的开关必须是簇中的最后一个开关。

### switch statement (switch语句)

这是一个程序技术，允许你计算一个表达式，并根据这个表达式的值完成一个多路分支，执行对应这个值的适当代码段。也称为“*case*结构”，这得名于类似的Pascal构造。Perl中大多数 *switch* 语句都拼作 *given*。参见第4章的“*given*语句”一节。

### symbol (符号)

一般地，这包括任何 *token* 或元符号。通常更特定地用来表示你在符号表中可能找到的某个名字。

### symbolic debugger (符号调试器)

这个程序允许你单步跟踪程序的执行，可以在某个地方停下来或打印状态，来查看是否有问题，如果确实出现问题，明确是什么问题。这里的“符号”是指你可以使用编写程序所用的同样的符号与调试器交流。

### symbolic link (符号链接)

一个候选的文件名，指向真正的文件名，文件名则指向真正的文件。操作系统要解析一个包含符号链接的路径名时，就会替换这个名，并继续解析。



### symbolic reference (符号引用)

这个变量的值是另外一个变量或子例程名。通过对第一个变量解引用,可以得到第二个变量。符号引用在`use strict "refs"`下是不合法的。

### symbol table (符号表)

编译器用符号表来记住符号。类似Perl的程序必须以某种方式记住你使用的所有变量、文件句柄和子例程的名字。这是通过把名字放在一个符号表中做到的,在Perl中,符号表使用一个散列表实现。对于各个包,有一个单独的符号表来为各个包提供它自己的命名空间。

### synchronous (同步)

这表示编程时事件序列的顺序可以确定;也就是说,事件会一个接一个发生,而不是同时发生。

### syntactic sugar (语法糖)

编写某个构造的一种更容易的候选方式;一种快捷方式。

### syntax (语法)

来自希腊语 $\sigma\acute{\upsilon}\nu\tau\alpha\chi\iota\varsigma$ ,表示“有安排”。事情(尤其是符号)相互之间如何组织。

### syntax tree (语法树)

程序的一种内部表示,在语法树中,底构造会挂在包围它们的高层构造下面。

### syscall (系统调用)

对操作系统的直接函数调用。你使用的很多重要的子例程和函数都不是直接的系统调用,而是建立在系统调用层之上的一层或多层。一般地,Perl程序员不需要担心这种区别。不过,如果你恰好知道Perl函数实际上是系统调用,就可以预测失败时它们中哪些函数会设置 $!(\$ERRNO)$ 变量。遗憾的是,初学的程序员通常错误地用“系统调用”来表示调用Perl的`system`函数,这实际上涉及到多个系统调用。为了避免这种混淆,我们几乎总是用“系统调用”来表示可以通过Perl的`syscall`函数间接调用的函数,而不是用Perl的`system`函数调用的函数。

### taint checks (污染检查)

Perl完成的特殊加工处理,用来跟踪程序中的外部数据流,并禁止在系统命令中使用这些外部数据。

### tainted (已污染)

是指数据来自用户的“脏手”,因此让一个安全程序依赖于这样的数据是不安全的。如果运行一个`setuid`(或`setgid`)程序,或者如果使用`-T`开关,Perl就会完成污染检查。

### taint mode (污染模式)

在`-T`开关下运行,将所有外部数据标志为可疑,并拒绝将其用于系统命令。另外参见第20章。

### TCP

传输控制协议(Transmission Control Protocol)的简写。这是包装在IP协议外的一个协议,使得不可靠的数据包传输机制对应用程序来说看起来是一个可靠的字节流(通常如此)。

### term (终点或项)

“终点”的简写。也就是说,这是语法树的一个叶子节点。在表达式中,对操作符来说,语法上相当于一个操作数的东西称为项。

### terminator (终止符)

标志另一个字符串结束的一个字符或字符串。 $\$/$ 变量包含终止一个`readline`操作的字符串,`chomp`会从字符串末尾删除这个终止符。不要与定界符或分隔符混淆。这个句子末尾的句号就是一个终止符。

### ternary (三元)

有3个操作数的操作符。有时拼读作`trinary`。

### text (文本)

一个字符串或文件,主要包含可打印的字符。

### thread (线程)

类似于一个派生的进程,不过没有`fork`固有的内存保护。线程比一个完整的进程更轻量级,因此一个进程可以有多个线程在其中运行,所有这些线程都争夺同一个进程的内存空间,除非采取措施来保护线程相互之间不会破坏。

### tie (绑定)

一个变量和它的实现类之间的纽带。参见第27章的`tie`函数和第14章。

### titlecase (标题形式)

使用这种形式表示大写字母后面是小写字



母而不是更多的大写字母。有时这也称为句子形式或标题形式。英语不使用Unicode标题形式，不过英语标题的大小写规则更为复杂，而不只是将每个单词的首字符大写。

#### TMTOWTDI

条条大路通罗马 (There's More Than One Way To Do It)，这是Perl的座右铭。这个概念是指语法树中可能有多个合法的路径可以解决当前上下文中的编程问题（这并不是说有更多方法就更好，或者所有可能的路径都同样必要，这只是说可能有不只一个可行的办法）。

#### token

编程语言中的一个字形，这是文本中有语义意义的最小单位。

#### tokenizer (词法分析器)

这个模块将程序文本分解为一个`token`序列，供解析器以后分析。

#### tokenizing (词法分析)

将程序文本分解为`token`。也称为“词法分析” (lexing)，在这种情况下则会得到“lexemes” (词素)，而不是`token`。

#### toolbox approach (工具箱方法)

这个概念是指，利用一个完备的简单工具集（而且这些工具可以很好地联合使用），你可以构建几乎你想构建的任何东西。如果你在组装一个三轮车，这就很好，不过如果你要构建一个消波交流镇流器 (defranchizing comboflux regurgalator)，可能希望有你自己的机器车间，能够构建特殊的工具。Perl就是这样一种机器车间。

#### topic (主题)

你处理的内容。类似`while(<>)`，`for`，`foreach`和`given`等结构都会通过为`$_`（默认`topic`变量）赋值来为你设置主题。

#### transliterate (变换)

通过将源字符串中的各个字符映射到结果字符串中相应的字符，从而将一个字符串表示转换为另外一种表示。不要与转换混淆：例如，希腊语`πολύχρωμος`变换为`polychromos`，但是转换为`many-colored`。另外参见第5章中的`tr///`操作符。

#### trigger (触发)

一个事件导致运行一个处理器。

#### trinary (三元)

不是一个有3个行星的恒星系，而是一个有3个操作数的操作符。有时拼读作`ternary`。

#### troff

一个令人尊敬的排版语言，Perl从中得到了`$$`变量的名字，另外在骆驼系列书的制作中都秘密使用了这个排版语言。

#### true

任何不计算为0或""的标量值。

#### truncating (截断)

清空一个文件现有的内容，可能是打开一个文件来完成写时自动截断，或者是显式通过`truncate`函数截断。

#### type (类型)

参见数据类型和类。

#### type casting (类型转换)

将数据从一个类型转换为另一种类型。C允许这样做。Perl不需要这么做，也不希望这么做。

#### typedef

C和C++语言中的类型定义。

#### typed lexical (有类型的词法作用域变量)

声明为有一个类类型的词法作用域变量：  
`my Pony $bill`。

#### typeglob (类型团)

使用有前缀\*的单个标识符。例如，`*name`表示任何或所有`$name`、`@name`、`%name`、`&name`，或者只是`name`。其用途（即如何使用）会确定如何解释这个类型，即解释为所有这些类型，还是只解释为其中一个类型。参见第2章中的“类型团和文件句柄”。

#### typemap (类型映射)

XS中编写的一个扩展模块中对C类型与Perl类型相互如何转换的描述。

#### UDP

用户数据报协议 (User Datagram Protocol)，在互联网上发送数据报的常用方法。

#### UID

用户ID。通常在文件上下文或进程所有权上下文中使用。

#### umask

权限位的一个掩码，创建文件或目录时应

当强制关闭这些权限位，从而建立一种策略来决定通常拒绝谁访问。参见umask函数。

### unary operator (一元操作符)

这是一个只有一个操作数的操作符，如!或chdir。一元操作符通常是前缀操作符，也就是说，它们在其操作数的前面。++和--操作符可以是前缀也可以是后缀（它们的位置会改变它们的含义）。

### Unicode

差不多涵盖世界上所有主要字符集的一个字符集。另外参见<http://www.unicode.org>。

### Unix

一个非常庞大而且在不断演进的语言，有多种候选的语法（大多不兼容）。利用Unix，任何人都可以选择他们自己的做法，而且通常确实是这样。使用这种语言的人认为它很容易学，因为可以很容易地针对个人自己的目的进行调整，不过方言上的差别会导致“种族”间的相互通信几乎是不可能的，所以人们通常将这个语言缩减为一个类似洋泾浜语的子集。要想让大家理解，Unix shell程序员必须花大量时间研究这门艺术。很多人已经放弃这个规程，现在更愿意通过一个类世界语的语言进行通信，这种语言就叫做Perl。在“古代”，Unix还用于表示贝尔实验室的一些人写的一些代码，来利用PDP-7计算机做一些工作（在当时这种计算机还做不了其他事情）。

### uppercase (大写)

在Unicode中，不只是通用类别为Uppercase Letter的字符，而是有Uppercase属性的所有字符，包括一些字母数字和符号。不要与标题形式混淆。

### value (值)

一段实际的数据，不同于所有变量、引用、键、索引、操作符以及诸如此类需要访问值的东西。

### variable (变量)

一个命名的存储位置，可以保存程序认为合适的任何类型的值。

### variable interpolation (变量内插)

将一个标量或数组变量内插到一个字符串。

### variadic (变参函数)

这是指一种乐于接收不定数目实参的函数。

### vector (向量)

表示一个标量值列表的数学用语。

### virtual (虚拟)

提供一种假象，但实际上并不是，如：虚拟内存并不是实际内存（参见内存）。与“虚拟”相对的是“透明”，表示提供实际的东西而不是表面现象，如：Perl透明地处理变长UTF-8字符编码。

### void context (void上下文)

一种标量上下文形式，其中表达式不会返回任何值，计算这个表达式只是为了得到其副作用。

### v-string

一个“版本”或“向量”字符串，用一个v后面跟一个十进制整数序列（采用点记法）来指定，如v1.20.300.4000。每个数会转换为有指定序数值的字符（如果至少有3个整数，v是可选的）。

### warning (警告)

打印到STDERR流的消息，表示某个地方可能出错但是不必为此完全崩溃。参见第27章的warn，以及第29章的warnings pragma。

### watch expression (监视表达式)

一个表达式的值改变时，会导致Perl调试器中出现一个断点。

### weak reference (弱引用)

通常并不计为引用的一个引用。数据的所有正常引用都消失时，这个数据也消失。这对于永远也不会消失的循环引用很有用。

### whitespace (空白符)

可以移动光标但是不会在屏幕上显示任何东西的一个字符。通常是指空格、tab、换行、回车或换页。在Unicode中，这会匹配很多其他的字符（Unicode认为它们是空白符），包括NO-BREAK SPACE。

### word (字)

在通常的“计算机术语”中，计算机处理这种大小的数据最为高效，这是2的一个幂，通常是32位。在Perl文化中，它更常指示一个字母数字标识符（包括下划线），

或者指示一个非空白符字符串的字符串，由空白符或字符串边界来定界。

#### working directory (工作目录)

你的当前目录，操作系统从这里开始解释相对路径名。操作系统知道你的当前目录，可能是因为你用chdir告诉过它，或者因为你从派生时父进程所在的位置开始。

#### wrapper (包装器)

这是一个程序或子例程，为你运行另外某个程序或子例程，会修改一些输入或输出从而更好地满足你的要求。

#### WYSIWYG

所见即所得。在屏幕上显示的东西与其实显示一致。另外表示与魔法相对，因为一切都像其显示的一样真实，如3参数形式的open。

#### XS

外部导入的一个非常优秀的外部子例程 (eXternal Subroutine)，用现有C或C++执行，或者由某种让人激动的扩展语言 (名为XS) 执行。

#### XSUB

XS中定义的一个外部子例程。

#### yacc

Yet Another Compiler Compiler。这是一个解析器生成器，如果没有它，Perl可能根本不会存在。参见Perl源代码发行版本中的文件perly.y。

#### zero width (0宽度)

这个子模式断言匹配字符间的null字符串。

#### zombie (僵尸)

一个进程已经死亡 (退出)，但是它的父进程还没有通过调用wait或waitpid得到通知。如果使用fork派生子进程，必须在子进程退出后进行清理，否则进程表会填满，你的系统管理员会对你很不满意。